

Optimización de flujo en redes

Portafolio

Mario Gutiérrez
1549273

3 de junio de 2019

Descripción

En el presente reporte se incluyen las distintas tareas realizadas a lo largo de la unidad de aprendizaje de optimización de flujo en redes (Enero-Junio 2019), además una breve descripción en base a la calificación obtenida.

Tarea 1

En esta práctica los principales errores fueron la falta de uso de ambiente matemático en las ecuaciones, el mal uso de números simples en párrafos, algunos errores de ortografía, la manera de referenciar las figuras, así como la bibliografía del documento.

Q.5

9273

Representación de redes a través de la teoría de grafos

(Mario Alberto Gutiérrez Carrales)

12 de febrero de 2019

En este reporte se desarrolla una investigación teórica acerca de las redes debido a que son de vital importancia ya que en el mundo se presenta una gran diversidad de problemas que pueden ser modelados a través de una red, de hecho, la existencia de las redes es tan relevante que la teoría de grafos [1] se encarga de estudiar sus propiedades.

Utilizando ésta teoría se pretende estructurar las propiedades de algunos grafos y además con la ayuda del lenguaje de programación interpretado **python** [2] se desean construir ejemplos que ayuden a visualizar de forma sencilla cada grafo.

En el código se utilizaron dos módulos, uno de ellos es NetworkX [3], que contiene estructura de datos que almacena grafos y el otro Matplotlib [4] que permite visualizar y guardar imágenes en formato eps.

El uso de dichos módulos va declarado al inicio del código, tal como se muestra a continuación:

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

Antes de comenzar con la descripción de varios tipos de grafos se presentan algunas definiciones preliminares que son necesarias.

Definiciones preliminares

Un grafo G es una tripleta $G=(V,E,\delta)$, donde V representa el conjunto de puntos llamados vértices (o nodos), E es el conjunto de arcos llamados aristas (o lados) y $\delta: E \rightarrow V \times V$ una función que asigna a cada arista $e \in E$ un par de vértices $u,v \in V$; en muchas ocasiones la función δ viene implícita en el dibujo del grafo.

Un grafo simple es aquel que entre cualquier pareja de vértices existe a lo más una arista que los une, en caso contrario se dice que es un multigrafo.

Se dice que un grafo es no dirigido si no se toma en cuenta el orden en que los aristas se asignan a los pares de vértices es decir $(v_i, v_j) = (v_j, v_i) \forall v_j, v_i \in V$, en caso contrario, es decir, si no se tiene la igualdad en alguna pareja de vértices se dice que es un grafo dirigido.

Si en un grafo se puede encontrar una lista de vértices conectados de tal manera que el primer vértice y el último sean el mismo entonces se dice que es un grafo cíclico, si un grafo no es cíclico entonces se dice que es acíclico.

Un grafo reflexivo es aquel en el que al menos un vertice tiene un arista que lo conecta consigo mismo.

Es fácil determinar el tipo de grafo que se tiene haciendo uso de la matriz de multiplicidad para multigrafos y adyacencia para grafos simples, por un lado si se desea saber si un grafo es dirigido basta con verificar que se cumpla que la matriz correspondiente sea simétrica, por otra parte para confirmar si un grafo es reflexivo es suficiente observar la diagonal principal de la matriz y verificar si hay un valor distinto de 0.

Es posible combinar varias propiedades, obteniendo así diferentes tipos de grafos como se muestran en las siguientes secciones.

NOTA: En los siguientes ejemplos se utilizan aristas azules para hacer referencia a que hay múltiples aristas entre 2 vértices. Se utilizan nodos verdes para indicar que un nodo es reflexivo y tiene un lazo, en cambio un nodo rojo no hace referencia a algo en especial.

1. Grafo simple no dirigido acíclico

Un modelo aplicado que se puede llevar a cabo con este tipo de grafos son las semifinales de un torneo de fútbol, tal como se muestra en la figura 1.

```
1 G=nx.Graph()
2 pos={0:(1,0),1:(2,0),2:(3,0),3:(4,0),4:(1.5,.8),5:(3.5,.8),
      6:(2.5,1.6)}
3 labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'7'}
4 Conexion=[(0,4),(1,4),(2,5),(3,5),(4,6),(5,6)]
5
6 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=range(7),
      node_color='r')
7 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=Conexion,width=1)
8 nx.draw_networkx_labels(G,pos,labels,font_size=12)
9 plt.axis('off')
10 plt.savefig("Ejemplo1.eps")
11 plt.show()
```

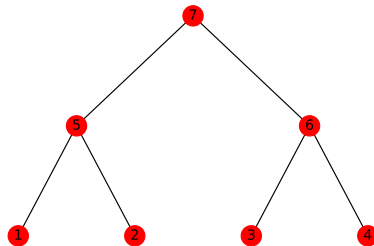


Figura 1: Seminifinales de torneo de fútbol

Cuando en este grafo hay una manera de llegar entre cada pareja de vértices se le conoce como árbol, por la forma parecida a un árbol real en como se van desprendiendo las aristas.

Otras aplicaciones de este tipo de grafo son los organigramas en general, distribución de bienes, árboles genealógicos, isómeros de cadena, etc.

2. Grafo simple no dirigido cíclico

Una de las aplicaciones más utilizadas hoy en día que se basan en este tipo de grafo es la representación de usuarios de una red social como Facebook [5], Twitter [6], etc. así como se observa en la figura 2.

```
1 G=nx.Graph()
2 pos={0:(2,1),1:(2.5,1),2:(1,2),3:(2.25,2),4:(3,2),5:(2,3),
      ,6:(2.5,3)}
3 labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'7'}
4 Conexion=[(0,2),(1,3),(2,3),(3,4),(2,5),(3,6),(4,6)]
5
6 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=range(7),
      node_color='r')
7 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=Conexion,width=1)
8 nx.draw_networkx_labels(G,pos,labels,font_size=12)
9 plt.axis('off')
10 plt.savefig("Ejemplo2.eps")
11 plt.show()
```

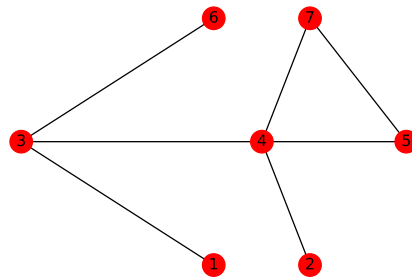


Figura 2: *Usuarios en una red social*

Otra aplicación importante es la modelación de territorios y de ahí surge uno de los problemas más importantes dentro de la optimización matemática que es el problema de la ruta más corta ~~(C~~ Considerando que todas las calles, avenidas ó carreteras son de doble sentido).

3. Grafo simple no dirigido reflexivo

Este grafo se encuentra presente en la forma en como están contruidos ciertos cruces de avenidas en diferentes ciudades, de tal manera que las avenidas son perpendiculares entre sí y en medio se encuentra un retorno, que representa un lazo en el grafo, y su objetivo es cambiar la orientación en la que se dirige el automovilista. ~~Observar~~ *Vease* figura 3.

```
1 G=nx.Graph()
2 pos={0:(0,1),1:(1,0),2:(1,1),3:(2,1),4:(1,2)}
3 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
4 Conexion=[(0,2),(1,2),(3,2),(4,2)]
5
6 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[2],
7     node_color='g')
8 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[0,1,3,4],
9     node_color='r')
10 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=Conexion,width=1)
11 nx.draw_networkx_labels(G,pos,labels,font_size=12)
12 plt.axis('off')
13 plt.savefig("Ejemplo3.eps")
14 plt.show()
```

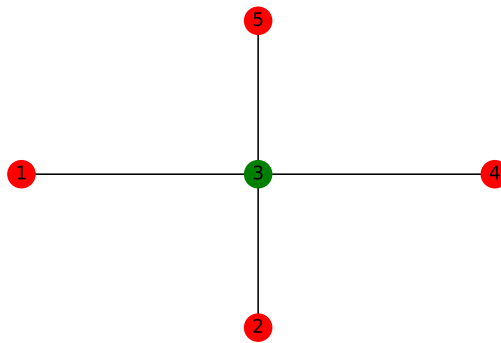


Figura 3: *Conexión entre avenidas y un retorno*

4. Grafo simple dirigido acíclico

Algunas de las principales aplicaciones de los grafos dirigidos es la modelación de redes de transporte, cadenas de suministro, organigramas con dirección, etc.

Supongamos que el dueño de una empresa quiere llevar a cabo un proyecto, el reparto de actividades se distribuye jerárquicamente como se muestra en la figura 4.

```
1 G=nx.DiGraph()
2
3 pos={0:(1.75,3),1:(.5,2),2:(3,2),3:(0,1),4:(1,1),5:(2.5,1),
4      ,6:(3.5,1)}
5 labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'7'}
6 Conexion=[(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)]
7 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=range(7),
8   node_color='r')
9 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=Conexion,width=1)
10 nx.draw_networkx_labels(G,pos,labels,font_size=12)
11 plt.axis('off')
12 plt.savefig("Ejemplo4.eps")
13 plt.show()
```

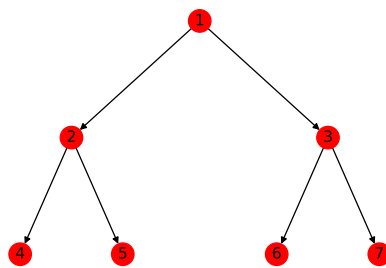


Figura 4: Jerarquía en planeación de un proyecto

5. Grafo simple dirigido cíclico

En muchas ocasiones el tráfico vehicular se ve afectado por diversos factores haciendo que este sea lento, debido a ello en muchas calles solamente se permite el acceso en un solo sentido. La figura 5 muestra un ejemplo del tránsito vehicular en una cierta localidad.

```
1 G=nx.DiGraph()
2
3 pos={0:(0,1),1:(1,0),2:(2,1),3:(1.6,2),4:(.5,2)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
5 Conexion=[(0,1),(1,2),(2,3),(2,4),(3,4),(4,0)]
6
7 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=range(5),
8     node_color='r')
9 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=Conexion,width=1)
10 nx.draw_networkx_labels(G,pos,labels,font_size=12)
11 plt.axis('off')
12 plt.savefig("Ejemplo5.eps")
13 plt.show()
```

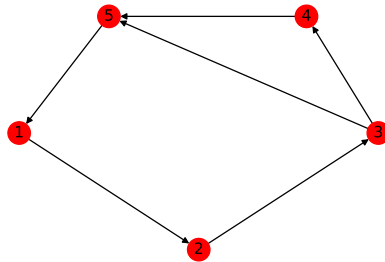


Figura 5: *Calles con sentido restringido*

6. Grafo simple dirigido reflexivo

En un modelo matemático se desea contemplar tantas características reales mientras sea posible para lograr entender un fenómeno, en este caso es posible mezclar la condición de un grafo dirigido que contenga un lazo tal como se observó en la sección 3, y un ejemplo de esto es el como están hechas ciertas colonias en algunos países en el que suele haber una calle en la esquina de una localidad y es acompañada de un retorno o parque para facilitar la salida vehicular. En la figura 6 se muestra un ejemplo de esta situación.

```
1 G=nx.DiGraph()
2
3 pos={0:(0,1),1:(1,0),2:(2,0),3:(2,1),4:(1,1)}
4 labels={0:'$1$',1:'2',2:'3',3:'4',4:'5'}
5 Conexion=[(0,1),(1,0),(1,2),(1,4),(2,3),(3,4),(4,3),(4,0)]
6
7 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[0,1,2,4],
8 node_color='r')
9 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[3],
10 node_color='g')
11
12 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=Conexion,width=1)
13 nx.draw_networkx_labels(G,pos,labels,font_size=12)
14
15 plt.axis('off')
16 plt.savefig("Ejemplo6.eps")
17 plt.show()
```

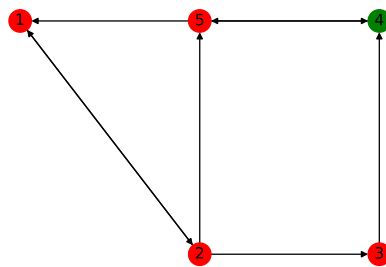


Figura 6: *Colonia*

7. Multigrafo no dirigido acíclico

En redes, un multigrafo es aquel que conecta algún par de vértices de múltiples maneras, esto hace que nuestro objeto de estudio sea más completo ya que en la mayoría de los casos esta herramienta es muy útil en la modelación. Un ejemplo de este grafo es una fábrica que produce productos y los suministra a diferentes almacenes pero los vehículos pueden transitar no solamente por una carretera, sino por múltiples, en este caso 2. Ver figura 7.

```
1 G=nx.MultiGraph()
2
3 pos={0:(1,.4),1:(2,.4),2:(2,0),3:(2,.8),4:(3.5,.8)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
5
6 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(1,2),(3,4)],
7                        width=1,edge_color='b')
8 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,1),(1,3)],
9                        width=1)
10
11 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=range(5),
12                        node_color='r')
13 nx.draw_networkx_labels(G,pos,labels,font_size=12)
14
15 plt.axis('off')
16 plt.savefig("Ejemplo7.eps")
17 plt.show()
```

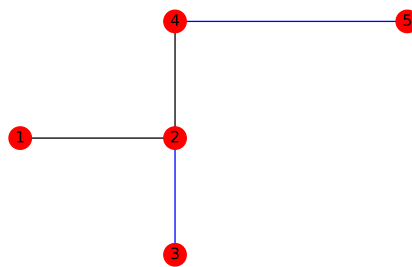


Figura 7: Ruta de un almacén

8. Multigrafo no dirigido cíclico

Este tipo de grafo se ve relacionado en general en la modelación donde se vean involucrados puentes. Una característica resaltante de este grafo es que se puede llegar a cualquier vértice por la conexión de las aristas. De ésto surge el problema particular de los 7 puentes de Königsberg. En la figura 8 se muestra un ejemplo.

```
1 G=nx.MultiGraph()
2
3 pos={0:(0,0),1:(2,0),2:(2,2),3:(0,2),4:(0,1)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
5
6 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,1),(1,2),(2,3)
7         ],width=1)
8 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,4),(3,4)],
9         width=1,edge_color='b')
10 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=range(5),
11         node_color='r')
12 nx.draw_networkx_labels(G,pos,labels,font_size=12)
13 plt.axis('off')
14 plt.savefig("Ejemplo8.eps")
15 plt.show()
```



Figura 8: *Puentes de una ciudad*

\section{...}
\label{mindar}

9. Multigrafo no dirigido reflexivo

\ref{...}

Sitúandonos en la sección ~~6~~, se puede extender el modelo de una colonia en la que la salida a una avenida puede estar dada por múltiples formas y no solo a la avenida, es decir, no solo un arista puede ser múltiple, sino varios. y además, puede considerarse múltiples lazos en la calle que funge el papel del nodo reflexivo, esto puede verse como un parque, callejón, retorno, etc. En la figura 9 se muestra un ejemplo.

```

1 G=nx.MultiGraph()
2
3 pos={0:(1,1),1:(0,0),2:(2,0),3:(2,2),4:(0,2)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
5
6 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,1),(0,2),(0,3),
7         (0,4),(1,4),(2,3),(3,4)],width=1)
8 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(1,2)],width=1,
9         edge_color='b')
10 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[0,1,2,3],
11         node_color='r')
12 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[4],
13         node_color='g')
14 nx.draw_networkx_labels(G,pos,labels,font_size=12)
15
16 plt.axis('off')
17 plt.savefig("Ejemplo9.eps")
18 plt.show()

```

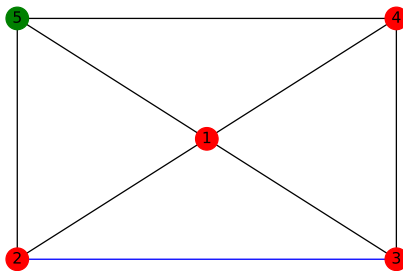


Figura 9: *Colonia con múltiples conexiones*

10. Multigrafo dirigido acíclico

Supongamos que se quiere llegar de una localidad a otra y se conocen los posibles caminos por los cuales se puede transitar y se desea encontrar la ruta que contenga menor costo para llevar a cabo el viaje. Dicho problema es el de la ruta más corta y este grafo puede modelarlo adecuadamente, así como se muestra en la figura 10.

```
1 G=nx.MultiDiGraph()
2
3 pos={0:(0,1),1:(1,0),2:(1,2),3:(2,0),4:(2,2)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
5
6 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,1),(1,3),(1,2),
7         (2,3),(2,4),(3,4)],width=1)
8 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,2)],width=1,
9         edge_color='b')
10 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist=[0,1,2,3,4],
11         node_color='r')
12 nx.draw_networkx_labels(G,pos,labels,font_size=12)
13 plt.axis('off')
14 plt.savefig("Ejemplo10.eps")
15 plt.show()
```

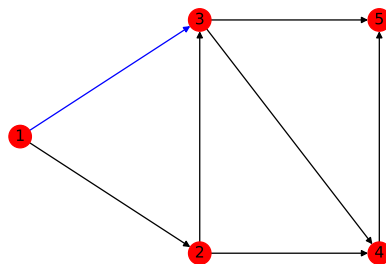


Figura 10: *Mapa transitorio*

11. Multigrafo dirigido cíclico

Este tipo de grafo es el que mejor modela los problemas de ruteo de vehículos, pues cuando estamos en alguna localidad siempre podemos llegar de nuevo a ella misma utilizando otras calles que, en ocasiones, se tiene la opción de elegir una entre varias calles para llegar a una localidad y también existe la posibilidad de que cualquier calle sea de un solo sentido. En la figura 11 se encuentra un ejemplo de este planteamiento.

```
1 G=nx.MultiDiGraph()
2
3 pos={0:(0,1),1:(1,0),2:(1,2),3:(2,0),4:(2,2),5:(3,1)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6'}
5
6 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,2),(1,0),(1,3),
7         (1,2),(2,3),(2,4),(3,1),(3,4),(3,5),(4,2),(4,5)],width=1)
8 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,1)],width=1,
9         edge_color='b')
10 nx.draw_networkx_nodes(G,pos,node_size=300,nodelist
11         =[0,1,2,3,4,5],node_color='r')
12
13 plt.axis('off')
14 plt.savefig("Ejemplo11.eps")
15 plt.show()
```

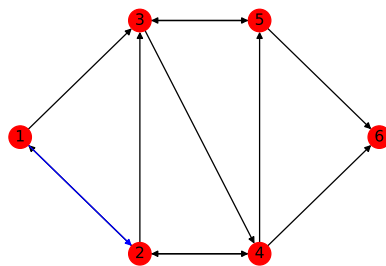


Figura 11: *Calles principales de una ciudad*

12. Multigrafo dirigido reflexivo

Un ejemplo aplicado de este tipo de grafo es la conectividad de cables y circuitos en una computadora, tanto los cables entrantes como por ejemplo los de encender y de internet, los circuitos mismos en el cpu, y los cables de salida como teclado, mouse, etc. Dicha representación de la red computacional se encuentra en la figura 12.

```
1 G=nx.MultiDiGraph()
2
3 pos={0:(0,2),1:(1,2),2:(1,1),3:(2,1),4:(2,2)}
4 labels={0:'1',1:'2',2:'3',3:'4',4:'5'}
5
6 nx.draw_networkx_nodes(G,pos,node_size=400,nodelist=[1],
7     node_color='g')
8 nx.draw_networkx_nodes(G,pos,node_size=400,nodelist=[0,2,3,4],
9     node_color='r')
10 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(0,1)],width=1,
11     edge_color='b')
12 nx.draw_networkx_edges(G,pos,alpha=1,edgelist=[(1,2),(1,3),(1,4)
13     ],width=1,edge_color='k')
14 nx.draw_networkx_labels(G,pos,labels,font_size=12)
15
16 plt.axis('off')
17 plt.savefig("Ejemplo12.eps")
18 plt.show()
```

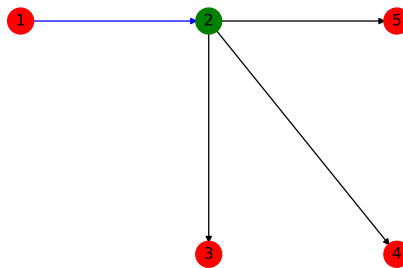


Figura 12: Red de conexión computacional

Referencias

- [1] John Michael Harris, Jeffrey L Hirst, and Michael J Mossinghoff. *Combinatorics and graph theory*, volume 2. Springer, 2008.
- [2] <https://www.python.org/>.
- [3] <https://networkx.github.io/documentation/networkx-1.10/reference/classes.html>.
- [4] <https://matplotlib.org/>.
- [5] <https://www.facebook.com/>.
- [6] <https://twitter.com>.

Tarea 2

En esta práctica los errores fueron breves y algunos de ellos fueron sobre identificar las palabras de otros idiomas y de ambiente computacional de otra tipografía, así como visualizar los nodos de los grafos de mayor tamaño.

Visualización de grafos

Mario Alberto Gutiérrez Carrales
9273

26 de febrero de 2019

Introducción

Este reporte está enfocado en la visualización de grafos mediante distintos algoritmos de acomodo que brinda `python` [1] en sus diferentes módulos.

Cuando recién se comienzan a usar los grafos que la librería `Networkx` ofrece, el usuario puede enfrentarse a situaciones como tener que posicionar los nodos para visualizar la representación de la red, justo como se observó en la tarea 1 [2]. Ante esta problemática y otras que pudieran existir, se proponen los algoritmos de acomodo (inglés: `layout`) que su objetivo es representar un grafo con ciertas características de la mejor manera visualmente posible, cabe destacar que en este caso se mostrará un algoritmo para cada ejemplo realizado en la práctica anterior.

Antes de comenzar con los ejemplos resulta interesante mencionar los aspectos computacionales que pudieran ser más relevantes o retadores en la codificación.

Aspectos computacionales

Se puede comenzar por describir las librerías que se utilizan para obtener exitosamente las representaciones de cada uno de los grafos, las cuales son [NetworkX](#) [3], [Matplotlib](#) [4] y [PyGraphviz](#) [5].

La primera proporciona los elementos que un grafo puede tener, como su estructura tanto de nodos como aristas. La segunda proporciona visualizaciones tanto en la consola como para guardar imágenes en el formato *eps* que fue el que se utilizó para la obtención de las figuras. Por último, cabe mencionar que la primer librería mencionada en este apartado a pesar de ser muy extensa, puede complementarse con algunas otras en el manejo de los grafos y especialmente en los algoritmos de acomodo, esta es una de las principales funciones de la tercer librería.

En cualquier lenguaje de programación se puede ver que hay partes repetitivas tales como librerías, impresiones, declaración de variables, etcétera y en python no es la excepción, así que con el objetivo de evitar fragmentos de códigos repetitivos se muestran inicialmente las partes que pueden presentarse con mayor frecuencia.

Código 1: Librerías usadas para la visualización de grafos

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import pygraphviz as pgv
```

Código 2: Quitar ejes y guardar el grafo en una imagen en formato eps

```
1 plt.axis('off')
2 plt.savefig("Ejemplo1.eps")
3 plt.show()
```

En el código 1 se muestran las librerías utilizadas y en el 2 como guardar una imagen en formato eps.

networkx
matplotlib
pygraphviz

Otro aspecto que más que repetitivo es tedioso, es cuando se pretende guardar una imagen, pero debido a las dimensiones del grafo se cortan los márgenes y se pierden información visual. Una manera de evitar esta situación está dada en el código 3.

Código 3: *Agrandar márgenes*

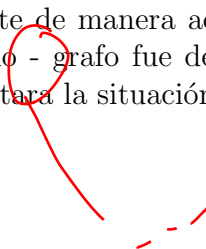
```
1 plot_margin = 0.05
2
3 x0, x1, y0, y1 = plt.axis()
4 plt.axis((x0 - plot_margin,
5           x1 + plot_margin,
6           y0 - plot_margin,
7           y1 + plot_margin))
```

Extraído de stackoverflow.com

En diversas versiones de python se presentan dificultades para trabajar con la librería pygraphviz como solución alterna se propone el uso de la versión web de colab de google, disponible en: <https://colab.research.google.com/notebooks/welcome.ipynb>

Para tratar de brindar la mayor uniformidad posible a las representaciones visuales, se utilizan arcos **negros** para indicar una arista simple, arcos **rojos** a aquellos aristas que contienen múltiples aristas, nodos **azules** que indican un nodo simple, y nodo **verde** para aquellos que tienen un lazo.

Se puede observar inicialmente que cada grafo puede ser representado visualmente de manera aceptable por más de un algoritmo, la elección entre algoritmo - grafo fue de acuerdo simplemente con que el dibujo realmente representara la situación de la que se está tratando.



1. Spectral

Este grafo representa las eliminatorias de un torneo de futbol. Se constan de 4 semifinalistas, de los cuales solamente 2 pasan a la siguiente fase y uno de ellos es quien queda campeón. El algoritmo que brinda una mejor representacion es el `spectral layout`, se puede programar como se muestra en el código 4 y el resultado de la visualización se muestra en la figura 1.

Código 4: *Spectral layout*

```
1 G = nx.Graph()
2
3 Conexion=[('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E'), ('C', 'F'), ('C', 'G')]
4 G.add_edges_from(Conexion)
5
6 labels={'A': 'C', 'B': 'F1', 'C': 'F2', 'D': 'SF1', 'E': 'SF2', 'F': 'SF4', 'G': 'SF3'}
7
8 pos = nx.spectral_layout(G)
9
10 nx.draw_networkx_nodes(G, pos, node_color='b', alpha=1, node_size=500)
11 nx.draw_networkx_edges(G, pos, edge_color='k', alpha=1, width=3)
12 nx.draw_networkx_labels(G, pos, labels, font_size=10, font_color='w')
```

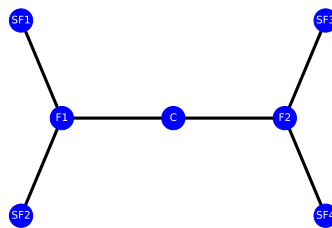


Figura 1: *Semifinales de torneo de futbol*

Se observa que la representación visual es bastante buena debido a la característica de árbol que esta situación exige, otro algoritmo que pudo haber sido considerado es el `bipartite layout`.

2. Bipartite

Una de las aplicaciones más utilizadas hoy en día es la representación de usuarios de una red social como Facebook, en ocasiones conviene saber dicha relación por género.

Código 5: *Bipartite layout*

```
1 G = nx.Graph()
2 G.add_nodes_from(range(6))
3 G.add_edges_from([(0,1),(1,2),(3,4),(1,5),(2,4),(4,5)])
4
5 pos = nx.bipartite_layout(G, {1,4})
6
7 labels={0: 'Ana', 1: 'Juan', 2: 'Paty', 3: 'Dany', 4: 'Jose', 5: 'Zoe'}
8
9 nx.draw_networkx_nodes(G, pos, node_color='b', alpha=1, node_size
    =600)
10 nx.draw_networkx_edges(G, pos, edge_color='k', alpha=1, width=3)
11 nx.draw_networkx_labels(G, pos, labels, font_size=10, font_color='w'
    )
```

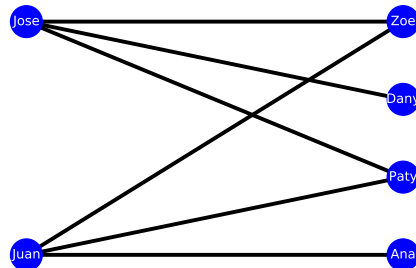


Figura 2: *Relación de amistad por género*

Como la situación está dada por género esto indica directamente a una partición del grafo en 2 secciones lo cual da la intuición que no hay mejor representación que la que se utilizó.

Los

2. Bipartite

Una de las aplicaciones más utilizadas hoy en día es la representación de usuarios de una red social como Facebook, en ocasiones conviene saber dicha relación por género.

Código 5: *Bipartite layout*

```
1 G = nx.Graph()
2 G.add_nodes_from(range(6))
3 G.add_edges_from([(0,1),(1,2),(3,4),(1,5),(2,4),(4,5)])
4
5 pos = nx.bipartite_layout(G, {1,4})
6
7 labels={0: 'Ana', 1: 'Juan', 2: 'Paty', 3: 'Dany', 4: 'Jose', 5: 'Zoe'}
8
9 nx.draw_networkx_nodes(G, pos, node_color='b', alpha=1, node_size
    =600)
10 nx.draw_networkx_edges(G, pos, edge_color='k', alpha=1, width=3)
11 nx.draw_networkx_labels(G, pos, labels, font_size=10, font_color='w'
    )
```

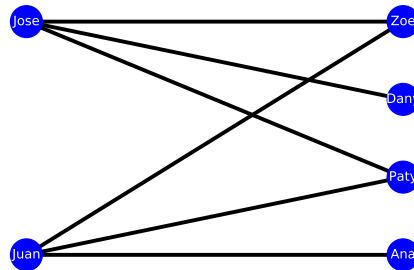


Figura 2: *Relación de amistad por género*

Como la situación está dada por género esto indica directamente a una partición del grafo en 2 secciones lo cual da la intuición que no hay mejor representación que la que se utilizó.

3. Random

Este ejemplo se basa en como están contruidos ciertos cruces de avenidas en diferentes ciudades, de tal manera que las avenidas son perpendiculares entre sí y en medio se encuentra un retorno, que representa un lazo en el grafo, y su objetivo es cambiar la orientación en la que se dirige el automovilista. Observar figura 3.

Código 6: *Random layout*

```
1 G = nx.Graph()
2
3 Conexion=[('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E')]
4 G.add_edges_from(Conexion)
5
6 labels={'A': 'Esq\n_#1', 'B': 'Retorno', 'C': 'Esq\n_#2', 'D': 'Esq\n_#3', 'E': 'Esq\n_#4'}
7
8 pos = nx.random_layout(G)
9
10 nx.draw_networkx_nodes(G, pos, node_size=1000, nodelist=['B'],
11                        node_color='g')
12 nx.draw_networkx_nodes(G, pos, node_size=1000, nodelist=['A', 'C', 'D', 'E'],
13                        node_color='b')
14 nx.draw_networkx_edges(G, pos, edge_color='k', alpha=1, width=3)
15 nx.draw_networkx_labels(G, pos, labels, font_size=8, font_color='w')
16 nx.draw_networkx_edge_labels(G, pos, edge_labels={('A', 'B'): 'Calle1', ('B', 'D'): 'Calle2', ('B', 'E'): 'Calle3', ('A', 'C'): 'Calle4'})
```

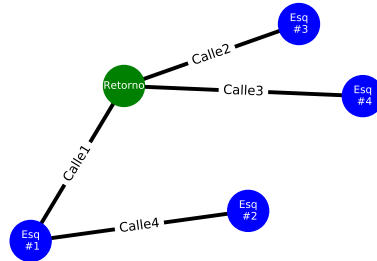


Figura 3: *Conexión entre avenidas y un retorno*

A pesar de que la representación es aceptable, no podemos decir que este algoritmo sea muy bueno, ya que así como un algoritmo random da una buena solución puede no darla, de hecho, si el código se corre varias veces, se obtienen diferentes versiones representativas, por lo que una vez hallado un dibujo que satisfaga al usuario, conserve la imagen con precaución.

4. Circular

En muchas ocasiones el tráfico vehicular se ve afectado por diversos factores haciendo que este sea lento, debido a ello en muchas calles solamente se permite el acceso en un solo sentido, así como se muestra en la figura 4.

Código 7: *Circular layout*

```
1 G = nx.DiGraph()
2 G.add_nodes_from(range(5))
3
4 pos = nx.circular_layout(G)
5
6 nx.draw_networkx_nodes(G, pos, node_size=600, nodelist=range(5),
7     node_color='b')
8 nx.draw_networkx_edges(G, pos, edgelist=[(0,1),(1,2),(2,3),(2,4),
9     (4,0)], edge_color='k', alpha=1, width=3, arrowsize=20)
10 nx.draw_networkx_labels(G, pos, {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'},
11     font_size=8, font_color='w')
12 nx.draw_networkx_edge_labels(G, pos, edge_labels={
13     (0,1): 'Calle1', (1,2): 'Calle2', (2,3): 'Calle3',
14     (2,4): 'Calle4', (4,0): 'Calle5'})
```

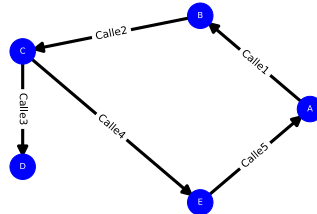


Figura 4: *Calles con sentido restringido*

La visualización que se obtuvo es realmente buena, ya que con la posición de los nodos y la introducción de etiquetas da una mejor representación de la realidad.

5. Pydot

Es importante modelar el como están hechas ciertas colonias en algunos países en el que suele haber una calle en la esquina de una localidad y es acompañada de un retorno o parque para facilitar la salida vehicular. En la figura 5 se muestra un ejemplo de esta situación.

Código 8: *Pydot layout*

```
1 G = nx.MultiDiGraph()
2 G.add_nodes_from(range(5))
3 G.add_edges_from([(0,1),(1,0),(1,2),(1,4),(2,3),(3,4),(4,0),
4                   ,(4,3)])
5 pos = nx.nx_pydot.pydot_layout(G)
6
7 nx.draw_networkx_nodes(G, pos, nodelist=[0,1,2,4], node_color='b',
8                        alpha=1, node_size=600)
9 nx.draw_networkx_nodes(G, pos, nodelist=[3], node_color='g', alpha
10                        =1, node_size=600)
11 nx.draw_networkx_edges(G, pos, edgelist=[(0,1),(1,0),(1,2),(1,4),
12                                           ,(2,3),(3,4),(4,0),(4,3)],
13                        edge_color='k', alpha=1, width=3,
14                        arrowsize=30)
15 nx.draw_networkx_labels(G, pos, labels={0: 'A', 1: 'B', 2: 'C', 3: 'D', 4:
16                                         'E'}, font_size=12, font_color='k')
```

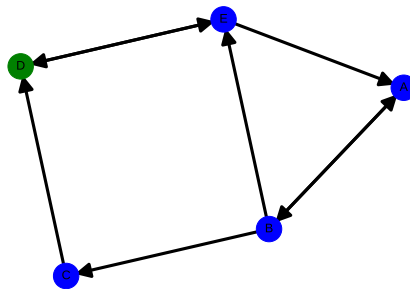


Figura 5: *Colonia*

6. Shell

Una fábrica que produce productos y los suministra a diferentes almacenes pero los vehículos pueden transitar no solamente por una carretera, sino por múltiples, en este caso 2. Ver figura 6.

Código 9: *Shell layout*

```
1 G = nx.MultiGraph()
2 G.add_nodes_from(range(5))
3
4 pos = nx.shell_layout(G)
5
6 nx.draw_networkx_nodes(G, pos, node_size=700, nodelist=[0],
7                        node_color='y', node_shape='s')
8 nx.draw_networkx_nodes(G, pos, node_size=700, nodelist=[1, 2, 3, 4],
9                        node_color='b')
10 nx.draw_networkx_edges(G, pos, edgelist=[(0, 1), (1, 3)], edge_color='k',
11                        alpha=1, width=3)
12 nx.draw_networkx_edges(G, pos, edgelist=[(1, 2), (3, 4)], edge_color='r',
13                        alpha=1, width=3)
14 nx.draw_networkx_labels(G, pos, {0: 'Dep', 1: 'A', 2: 'Suc_\n#1', 3: 'B',
15                                4: 'Suc_\n#2'}, font_size=8, font_color='w')
```

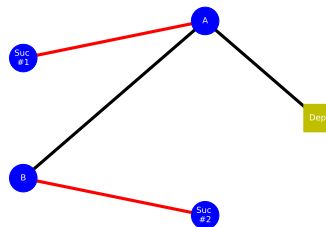


Figura 6: *Ruta de un almacén*

Este tipo de situación es de alta exigencia ya que en la práctica no se cuentan con pocos nodos, sino una gran cantidad y algo que puede considerarse es sustituir los nodos por las fotografías de las sucursales.

7. Fruchterman reingold

Un problema interesante en la práctica es donde se ven involucrados puentes. Una característica resaltante de este grafo es que se puede llegar de un vértice a cualquier otro por la conexión de las aristas.

Código 10: *Fruchterman reingold layout*

```
1 G=nx.MultiDiGraph()
2
3 G.add_nodes_from(range(5))
4 pos = nx.fruchterman_reingold_layout(G)
5
6 nx.draw_networkx_nodes(G, pos, node_size=500, nodelist=[0],
7                        node_color='g')
8 nx.draw_networkx_nodes(G, pos, node_size=500, nodelist=[1,2,3,4],
9                        node_color='b')
10 nx.draw_networkx_edges(G, pos, edgelist=[(1,2),(2,1),(2,3),(3,2),
11                                           (4,0)], edge_color='k', alpha=1, width=3, arrowsize=20)
12 nx.draw_networkx_edges(G, pos, edgelist=[(3,4),(0,1),(1,0)],
13                           edge_color='r', alpha=1, width=3)
14 nx.draw_networkx_labels(G, pos, {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'},
15                           font_size=8, font_color='w')
```

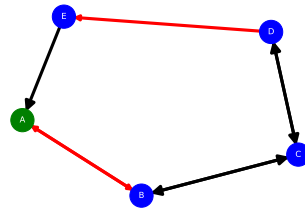


Figura 7: *Puentes de una ciudad*

Difícilmente los puentes tienen esta estructura, más bien son paralelos entre sí, pero esta representación puede ayudar en el caso en que se desea realizar un análisis para algún recorrido (como el problema de los ⁷¹ puentes).

file

8. Graphviz

Supóngase que en una colonia privada se tiene la libertad de transitar libremente por cualquier calle, donde al fondo hay un parque y este está en una calle que empieza y termina donde mismo, y además se puede obtener acceso a la avenida principal mediante dos calles. La figura 8 representa esta situación.

Código 11: *Graphviz layout*

```
1 G = nx.Graph()
2 G.add_nodes_from(range(5))
3 G.add_edges_from([(0,1),(0,2),(0,3),(0,4),(1,2),(1,4),(2,3),
4                   ,(3,4),])
5 pos = nx.nx_pydot.graphviz_layout(G)
6
7 nx.draw_networkx_nodes(G, pos, nodelist=range(4), node_color='b',
8                       alpha=1, node_size=600)
9 nx.draw_networkx_nodes(G, pos, nodelist=[4], node_color='g', alpha
10                       =1, node_size=600)
11 nx.draw_networkx_edges(G, pos, edgelist=[(0,1),(0,2),(0,3),(0,4),
12                                           ,(1,4),(2,3),(3,4)], edge_color='k', alpha=1, width=3)
13 nx.draw_networkx_edges(G, pos, edgelist=[(1,2)], edge_color='r',
14                       alpha=1, width=3)
15 nx.draw_networkx_labels(G, pos, labels={0: 'A', 1: 'B', 2: 'C', 3: 'D', 4:
16                                         'E'}, font_size=12, font_color='k')
```

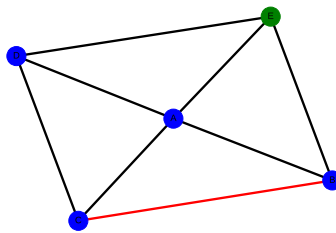


Figura 8: *Colonia con múltiples conexiones*

En la práctica cada colonia tiene su forma particular, pero esta representación es suficiente para lograr percibir la idea de la estructura de muchas colonias, así que para este ejemplo basta con esta representación.

9. Kamada kawai

Se desea llegar de una localidad a otra y se conocen los posibles caminos por los cuales se puede transitar y se desea encontrar la ruta que contenga menor costo para llevar a cabo el viaje.

Código 12: *Kamada kawai layout*

```
1 G = nx.MultiDiGraph()
2 G.add_nodes_from(range(5))
3
4 pos = nx.kamada_kawai_layout(G)
5
6 nx.draw_networkx_nodes(G, pos, node_size=400, nodelist=range(5),
7     node_color='b')
8 nx.draw_networkx_edges(G, pos, edgelist=[(1,2),(1,3),(3,4)],
9     edge_color='r', alpha=1, width=3)
10 nx.draw_networkx_edges(G, pos, edgelist=[(0,1),(0,2),(2,3),(2,4)],
11     edge_color='k', alpha=1, width=3)
12 nx.draw_networkx_labels(G, pos, labels={0: 'A', 1: 'B', 2: 'C', 3: 'D', 4:
13     'E'}, font_size=12, font_color='w')
```

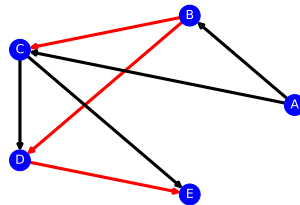


Figura 9: *Mapa transitorio*

La función de los grafos que representan redes de carreteras son más buenos por la información con la que pueden contar como etiquetas o pesos que por la visualización en sí, así que casi cualquier algoritmo de acomodo podría ser bueno para esta situación. Para este tipo de problema se puede proponer como trabajo a futuro el representar las carreteras sobre mapas.

10. Spring

Los problemas de ruteo de vehículos son de gran relevancia pues tienen muchas aplicaciones, dado que se puede modelar mediante un multigrafo dirigido ciclico por sus características físicas, se puede modelar como se muestra en la figura 10.

Código 13: *Spring layout*

```

1 G=nx.MultiDiGraph()
2 G.add_nodes_from(range(6))
3
4 pos = nx.spring_layout(G, iterations=300)
5
6 nx.draw_networkx_edges(G, pos, alpha=1, edgelist=[(0,2),(1,3),(1,2),
7             (2,3),(2,4),(3,1),(3,4),(3,5),(4,2),(4,5)], width=1, arrowsize
8             =20)
9 nx.draw_networkx_edges(G, pos, alpha=1, edgelist=[(0,1),(1,0)],
10             width=1, edge_color='r', arrowsize=20)
11 nx.draw_networkx_nodes(G, pos, node_size=400, nodelist
12             =[0,1,2,3,4,5], node_color='b')
13 nx.draw_networkx_labels(G, pos, {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'},
14             font_size=8, font_color='w')

```

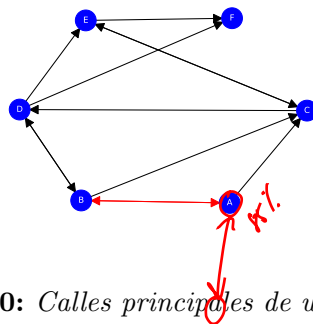


Figura 10: *Calles principales de una ciudad*

El algoritmo empleado para este grafo se aplica muy bien ya que trata de esparcir todos los nodos de forma equitativa dentro de un área, y dependiendo del número de nodos.

11. Force atlas 2

La conectividad de cables y circuitos en una computadora, tanto los cables entrantes como por ejemplo los de encender y de internet, los circuitos mismos en el cpu, y los cables de salida como teclado, mouse, etc. Dicha representación de la red computacional se encuentra en la figura 11.

Código 14: *Force atlas layout*

```
1 forceatlas2 = ForceAtlas2(  
2     outboundAttractionDistribution=True,  
3     linLogMode=False,  
4     adjustSizes=False,  
5     edgeWeightInfluence=1.0,  
6  
7     jitterTolerance=1.0,  
8     barnesHutOptimize=True,  
9     barnesHutTheta=1.2,  
10    multiThreaded=False,  
11  
12    scalingRatio=2.0,  
13    strongGravityMode=False,  
14    gravity=1.0,  
15  
16    verbose=True)  
17  
18 positions = forceatlas2.forceatlas2_networkx_layout(G, pos=None,  
    iterations=200)
```

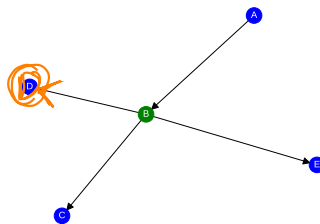


Figura 11: *Red de conexión computacional*

Este algoritmo proporciona una buena representación, el único detalle es que las aristas están muy largas, provocando que el grafo sea extenso si se empiezan a considerar mayor cantidad de nodos.

Referencias

- [1] Python. <https://www.python.org/>.
- [2] ~~G. Mario~~. <https://github.com/MarioGtz14/FlujoRedesMario>.
- [3] <https://networkx.github.io/documentation/networkx1.10/reference/classes.html>.
- [4] Matplotlib. <https://matplotlib.org/>.
- [5] PyGraphviz. <https://pygraphviz.github.io/documentation/latest/tutorial.html>.

Tarea 3

En esta práctica hubo errores de ortografía, así como gráficos con falta de fundamento matemático para completar las conclusiones a lo que se pedía.



Medición de tiempo de ejecución

Mario Alberto Gutiérrez Carrales
1549273

19 de marzo de 2019

Introducción

Esta práctica está enfocada en los algoritmos de optimización que la librería **NetworkX** [1] ofrece. En el repositorio de dicha librería se encuentra disponible documentación gratuita que ayuda al usuario a manipular grafos, creándolos, ubicándolos y no menos importante optimizando problemas de flujo en redes en ellos.

Cuando se aplica un algoritmo a un problema de optimización es importante conocer varios aspectos que permiten saber el funcionamiento de dicho algoritmo. El principal aspecto de interés es saber si se brinda la solución óptima, y después de eso, el tiempo que le toma al algoritmo para obtenerla, a este tiempo se le conoce como tiempo de ejecución.

En la versión 2.2 de **NetworkX** se encuentran más de 50 algoritmos, de los cuales, en esta práctica se trabajará con 5 de ellos, dándose a conocer en las siguientes secciones.

Aspectos computacionales

Para llevar a cabo la experimentación se utiliza `python` [2] y para poder utilizar las funciones requeridas se necesitan las siguientes librerías:

1. **NetworkX**. Como se mencionó anteriormente, esta librería es de gran ayuda para la manipulación de grafos.
2. **Matplotlib** [3]. Esta librería contiene funciones que sirven para visualizar gráficas y guardar imágenes en distintos formatos, en particular el `eps`.
3. **Time** [4]. Contiene la función que determina el tiempo (en segundos) que uno o varios procesos tardan, dicho tiempo se calcula con una diferencia entre el tiempo inicial y el final.
4. **Numpy** [5]. Entre las funciones que proporciona esta librería, se encuentra aquella que calcula la media y desviación estándar de una lista, en este caso será necesaria para calcular tiempos promedios y desviaciones estándar entre los tiempos de algoritmos.

En el código 1 se observa como se mandan llamar las librerías dentro del editor de texto, en el 2 como se guardan las imágenes en formato `eps` y en el 3 como se hacen más grandes los bordes de la imagen para evitar que se corten los márgenes de la imagen.

Código 1: *Librerías usadas para la experimentación*

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import time
4 import numpy as np
```

Código 2: *Guardar imágenes en formato eps*

```
1 plt.savefig("Corrida" + str(contador) + ".eps")
```

Código 3: *Aumentar bordes para evitar cortes en el grafo*

```
1 plot_margin = 0.05
2
3 x0, x1, y0, y1 = plt.axis()
4
5 plt.axis((x0 - plot_margin,
6           x1 + plot_margin,
7           y0 - plot_margin,
8           y1 + plot_margin))
```

El objetivo de la experimentación es realizar una comparación entre los tiempos que un algoritmo puede tardar para resolver un problema en distintas redes, para eso, se utiliza una laptop con sistema operativo de 64 bits y un procesador AMD A9-9410 RADEON R5, 5 COMPUTE CORES 2C+3G 2.90 GHz.

Sc

Dado que los algoritmos cuentan con una eficiencia muy potente en cuanto a tiempo cuando se resuelve una instancia pequeña, se recurre a utilizar 75,000 repeticiones y así obtener tiempos no nulos.

La metodología a seguir para cada algoritmo es la siguiente: se seleccionan 5 algoritmos y 5 grafos, los grafos son extraídos del repositorio de Gutiérrez [6], de las tareas 1 y 2 en donde cada algoritmo de solución se aplica a los 5 grafos, mostrando el histograma de comparación de tiempos para cada grafo. Cabe destacar que cada tipo de algoritmo funciona sobre distintos grafos, ya sea dirigidos, no dirigidos o ambos, así como ponderados o no llevando a que se realicen las modificaciones necesarias para poder cumplir con los requerimientos del algoritmo.

1. *A*ll shortest paths

Este algoritmo de optimización resuelve el problema de la ruta más corta, recibe como parámetros un grafo que puede contar con las condiciones del cuadro 1, un nodo fuente y un nodo destino, el atributo correspondiente al peso de los arcos y finalmente el algoritmo de solución que por default se considera el algoritmo de Dijkstra.

Especificaciones

Cuadro 1: Condiciones del grafo para el algoritmo 1.

Dirigido	No dirigido	Ponderado	No ponderado
Si	Si	Si	Si*

*Si no está ponderado, el algoritmo considera cada arista con costo igual a uno.

En el código 4 se muestra como se emplea este algoritmo.

Código 4: Algoritmo *all shortest paths*

```
1 Algoritmo = nx.all_shortest_paths(G1, source=0,  
target=4, weight = 'peso')
```

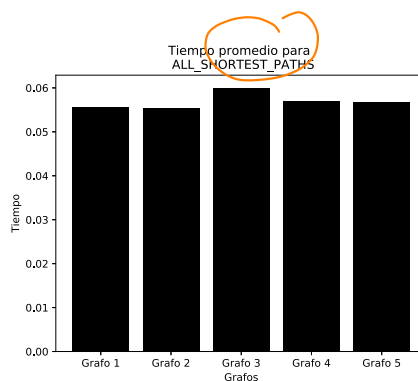
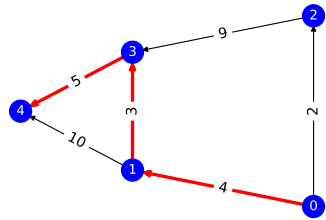
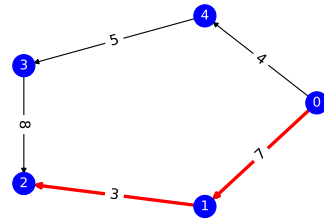


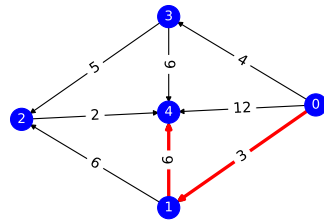
Figura 1: Histograma de los tiempos promedio por grafo



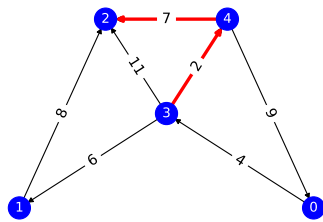
(a) Grafo 1



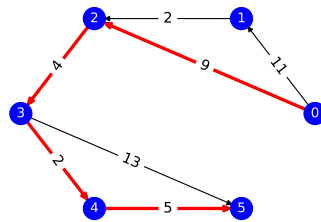
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 2: Solución de la ruta más corta a los grafos.

2. ~~betweenness centrality~~

Este algoritmo calcula la centralidad de intermediación de ruta más corta para los nodos. Recibe como parámetros un grafo que puede contar con las condiciones del cuadro 2 y si se desea normalizar el coeficiente o no (True por defecto).

Especificaciones

Cuadro 2: *Condiciones del grafo para el algoritmo 2.*

Dirigido	No dirigido	Ponderado	No ponderado
Si	Si	Si*	Si*

*Si no se pondera, se consideran todos los arcos con el mismo peso.

En el código 5 se muestra como se emplea este algoritmo.

Código 5: *Algoritmo betweenness centrality*

```
1 Coeficientes = nx.betweenness centrality (G6,  
normalized=True)
```

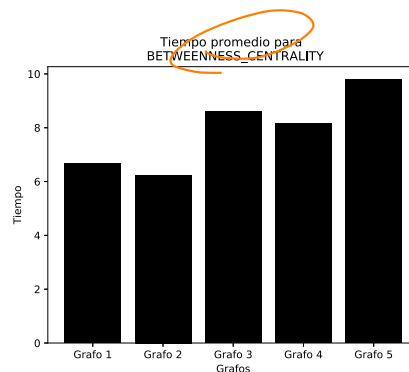
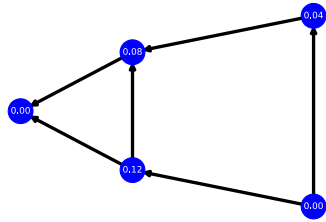
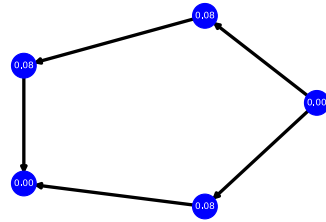


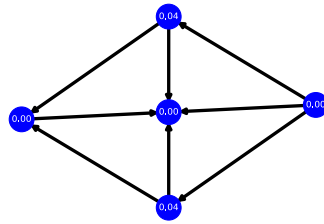
Figura 3: *Histograma de los tiempos promedio por grafo*



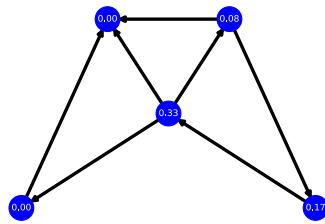
(a) Grafo 1



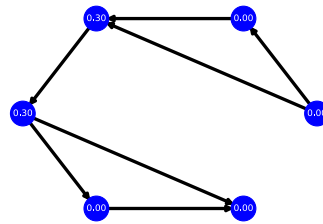
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 4: *Solución del coeficiente de centralidad intermedia.*

3. coloring greedy color

Este algoritmo da solución al problema de coloreo de grafos, el cual trata de colorear todos los nodos de un color que para cada par adyacente de nodos no se puede tener el mismo color, siendo así, la función retorna los nodos del grafo y el color que posee utilizando diversas estrategias.

Especificaciones

Cuadro 3: *Condiciones del grafo para el algoritmo 3.*

Dirigido	No dirigido	Ponderado	No ponderado
Si	Si	Si*	Si*

*Es indiferente si está ponderado o no, ya que este algoritmo se enfoca en los nodos.

En el código 6 se muestra como se emplea este algoritmo.

Código 6: *Algoritmo coloring greedy color*

```
1 d = nx.coloring.greedy_color(G11, strategy='largest_first')
```

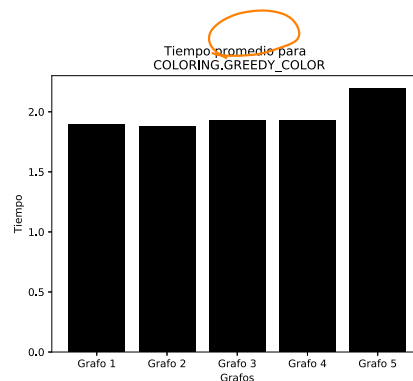
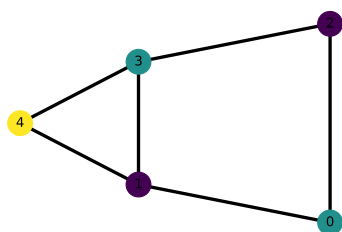
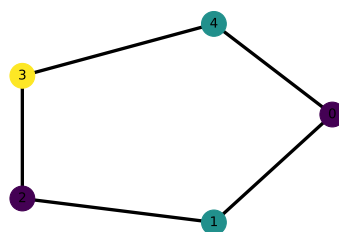


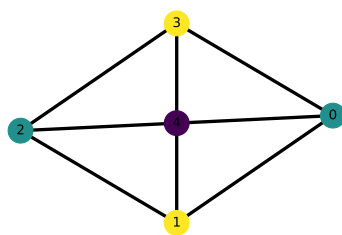
Figura 5: *Histograma de los tiempos promedio por grafo*



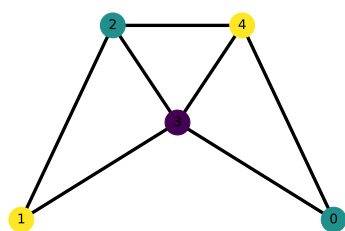
(a) Grafo 1



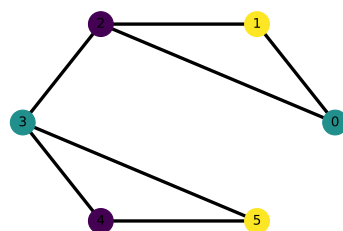
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 6: *Solución al coloreo de grafos.*

4. *M*aximum flow

Este algoritmo resuelve el problema de máximo flujo en una red, esta función toma como parámetros un grafo con las características mostradas en el cuadro 4, un nodo origen y otro destino, la capacidad máxima entre cada arco y retorna el valor de máximo flujo y también cual es la cantidad que fluye por cada arco.

Especificaciones

Cuadro 4: Condiciones del grafo para el algoritmo 4.

Dirigido	No dirigido	Ponderado	No ponderado
Si	Si	Si	Si*

*Si un arco no está ponderado, lo toma como si tuviera capacidad infinita.

En el código 7 se muestra como se emplea este algoritmo.

Código 7: Algoritmo maximum flow

```
1 Max_Flujo , Flujo_Arcos = nx.maximum_flow(G16, 0,4 ,  
    capacity='capacidad')
```

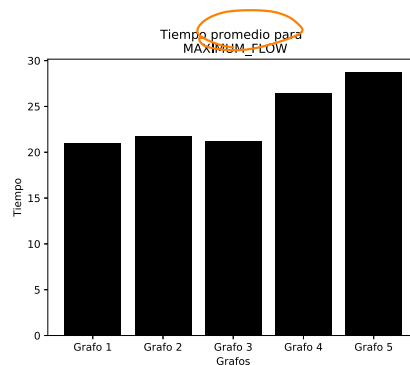
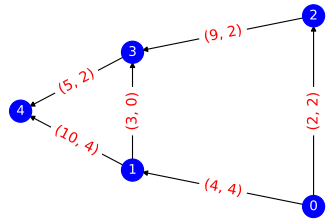
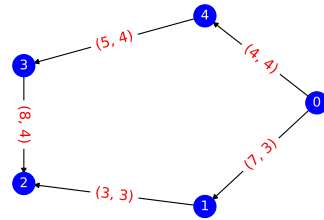


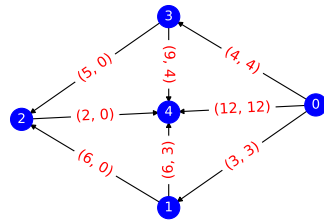
Figura 7: Histograma de los tiempos promedio por grafo



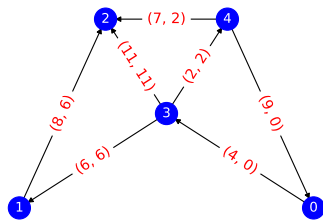
(a) Grafo 1



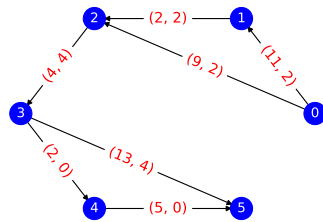
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 8: Solución del máximo flujo en una red.

5. *M*inimum spanning tree

Este algoritmo resuelve el problema del árbol de expansión mínima, retorna arcos del grafo original que cubre todos los nodos y no se pueden formar ciclo y es el de menor costo posible. La función toma como parámetros un grafo G con condiciones mostradas en el cuadro 5, pesos (opcional) y un algoritmo de solución deseado, por default se considera el algoritmo de Kruskal. Especificaciones

Cuadro 5: Condiciones del grafo para el algoritmo 5.

Dirigido	No dirigido	Ponderado	No ponderado
Sí	Sí	Sí	Sí*

*Si el grafo es no ponderado considera el peso de los arcos como uno.

En el código 8 se muestra como se emplea este algoritmo.

Código 8: Algoritmo *minimum spanning tree*

```
1 Tm = nx.minimum_spanning_tree(G21, algorithm='kruskal')
```

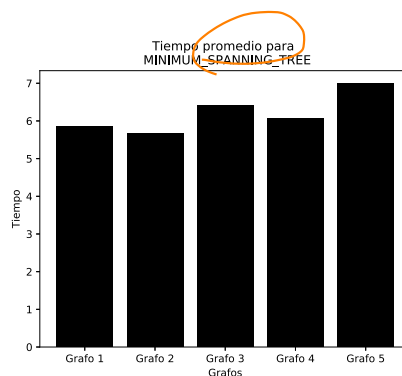
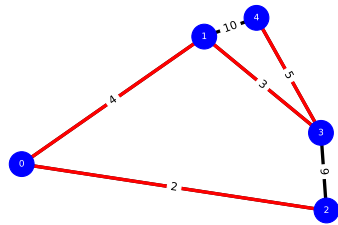
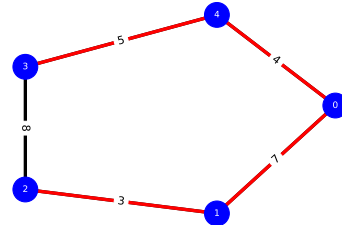


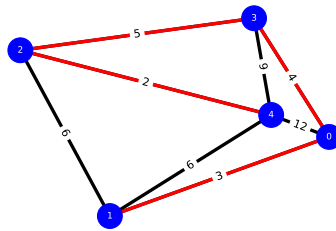
Figura 9: Histograma de los tiempos promedio por grafo



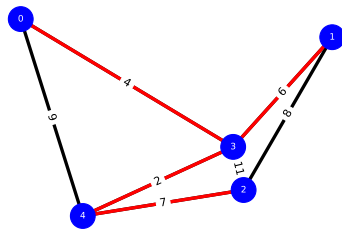
(a) Grafo 1



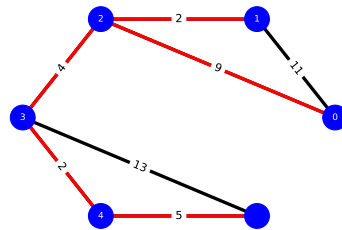
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 10: Solución del árbol de expansión mínima.

6. Comparación de algoritmos

En este último apartado se presenta un grafico que resuma de manera breve como es el comportamiento de los algoritmos siendo el que consume mayor tiempo el algoritmo 4, luego entre el 2 y el 5, posteriormente el 3 y al final el 1.

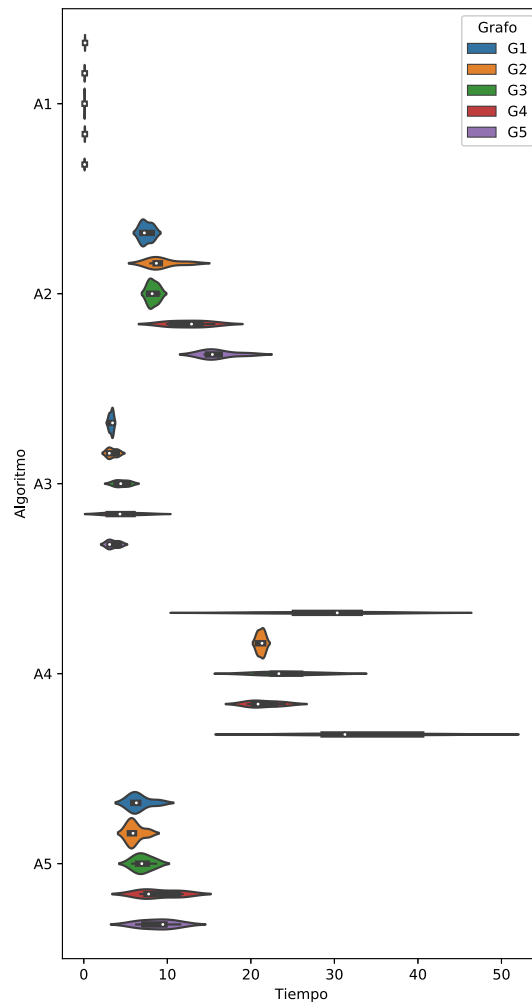


Figura 11: Grafica de violín para los algoritmos y grafos

Referencias

- [1] NetworkX developers Versión 2.2. <https://networkx.github.io>.
- [2] Python Software Foundation Versión 3.7.2. <https://www.python.org/>.
- [3] The Matplotlib development team Versión 3.0.2. <https://matplotlib.org/>.
- [4] Sulce A. PythonHow. <https://pythonhow.com/measure-execution-time-python-code/>.
- [5] NumPy developers Versión 1.16.2. <https://networkx.github.io>.
- [6] Gutiérrez M. Repositorio optimización flujo en redes. https://github.com/MarioGtz14/Flujo_Redes_Mario.

Tarea 4

En esta práctica los errores fueron en referirse a vértices y nodos de manera espontánea sin emplear un término fijo, así como arista y arco, también hubo algunos errores en el uso de ambiente matemático en las ecuaciones.



Complejidad asintótica experimental

Mario Alberto Gutiérrez Carrales
1549273

2 de abril de 2019

Introducción

En esta práctica el enfoque central son los algoritmos de generación de grafos y las implementaciones de algoritmos de flujo máximo, utilizando la librería de NetworkX [1].

Se plantean dos fases, la primera consiste en generar una matriz de datos de tiempos que representan el tiempo requerido que toma al algoritmo de solución resolver el grafo generado por el algoritmo de generación para esto se consideran algunos factores como réplicas de grafos, el número de nodos del grafo y la pareja de nodos fuente-destino.

La segunda fase consiste en realizar un análisis de varianza de un modelo donde el tiempo es la variable dependiente y las independientes son el algoritmo generador, algoritmo de solución, orden logarítmico y la densidad del grafo con el objetivo de saber efectos simples o de interacción son estadísticamente significativos.

1. Algoritmos

Generadores de grafos. Cada generador crea un grafo dependiendo los parámetros que este recibe, es común que reciban un parámetro omitiendo los demás donde dicho parámetro indica el número de nodos, aunque no en todos los casos es así. En los siguientes puntos se habla a detalle de los generadores que se estudian en esta práctica.

Grid graph. Genera una ~~ma~~^{§§}lla n -dimensional rectangular, se especifica al generador la dimensión que se desea y el número de nodos para cada componente. En el código 7 se muestra como se utiliza la función y los parámetros que necesita para un caso en donde la dimensión es dos y de forma cuadrada, al dar el parámetro n crea un grafo de n^2 nodos y $2(n^2 - n)$ aristas.

Código 1: *Ejemplo generador Grid_graph*

```
1 G = nx.grid_graph(dim=[x1, x1])
```

Complete graph. Este generador crea un grafo muy utilizado en la práctica que es el grafo completo, que es aquel en el que para cada par de vértices hay un camino que los une. La función tiene como parámetro el número de nodos que se desea que tenga el grafo. En el código 2 se muestra un ejemplo de esto. Si a este generador se le da como parámetro un entero n , entonces crea un grafo con n nodos y $c(n, 2)$ aristas.

Código 2: *Ejemplo generador Complete_graph*

```
1 H = nx.complete_graph(x1)
```

Circular ladder graph. Al aplicar este generador se produce un grafo que consiste en dos ciclos de ~~n~~^{§§} nodos en donde cada uno de los ~~n~~^{§§} pares de nodos se unen por una arista. La función toma como parámetro un entero n y crea un grafo con ~~2n~~^{§§} nodos y ~~3n~~^{§§} aristas. En el código 3 se muestra como se lleva a cabo el uso de este generador.

Código 3: *Ejemplo generador Circular_ladder_graph*

```
1 W= nx.circular_ladder_graph(x1)
```

En la figura 1 se muestra un ejemplo de las visualizaciones para cada generador utilizado en esta práctica.

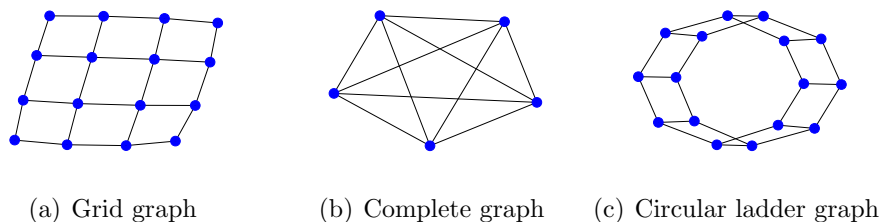


Figura 1: Visualización de grafos creados con los algoritmos generadores.

Implementaciones para el problema de máximo flujo. El problema de máximo flujo es un problema muy interesante en la optimización en redes, debido a ello se han desarrollado un gran número de implementaciones en diferentes softwares, en NetworkX hay una lista con una gran cantidad de implementaciones. En los siguientes puntos se describen aquellas que se utilizarán en esta práctica.

Maximum flow. Esta implementación retorna el valor de máximo flujo y además la cantidad que fluye por cada arco. En el código 4 se muestra un ejemplo de esta implementación.

Código 4: Ejemplo generador *Grid_graph*

```
1 Max_Flujo, Flujo_Arcos = nx.maximum_flow(G, (origen1, origen2), (destino1, destino2), capacity='weight')
```

Minimum cut value. El valor de mínimo corte es retornado al llevar a cabo esta implementación. En el código 5 se muestra como se utiliza esta implementación.

Código 5: Ejemplo generador *Grid_graph*

```
1 cut_value = nx.minimum_cut_value(G, (origen1, origen2), (destino1, destino2), capacity='weight')
```

Maximum flow value. Se retorna solamente el valor de máximo flujo. Se muestra un ejemplo en el código 6.

Código 6: Ejemplo generador Grid_graph

```
1 flow_value = nx.maximum_flow_value(G, (origen1 ,
    origen2) ,(destino1 , destino2) , capacity='weight')
```

En los tres casos la función recibe como parámetros el nodo fuente y destino, así como el atributo que tiene el papel de la capacidad. En las implementaciones para el problema de máximo flujo, si un arco no tiene una capacidad asignada se considera como infinita.

Se puede tener acceso a todos los algoritmos generadores de grafos y las implementaciones para el problema de máximo flujo a través de los siguientes

links: enlaces

Generadores.

<https://networkx.github.io/documentation/stable/reference/generators.html>

Implementaciones.

<https://networkx.github.io/documentation/stable/reference/algorithms/flow.html#module-networkx.algorithms.flow>

2. Ambiente computacional

Para llevar a cabo la experimentación se utiliza `python` [2] y para poder utilizar las funciones requeridas se necesitan las siguientes librerías:

1. `NetworkX`. Como se mencionó anteriormente, esta librería es de gran ayuda para la manipulación de grafos.
2. `Matplotlib` [3]. Esta librería contiene funciones que sirven para visualizar gráficas y guardar imágenes en distintos formatos, en particular el `eps`. EPS
↓
tex+tt
3. `Time` [4]. Contiene la función que determina el tiempo (en segundos) que uno o varios procesos tardan, dicho tiempo se calcula con una diferencia entre el tiempo inicial y el final.
4. `Numpy` [5]. Entre las funciones que proporciona esta librería, se encuentra aquella que calcula la media y desviación estándar de una lista, en este caso será necesaria para calcular tiempos promedios y desviaciones estándar entre los tiempos de algoritmos.
5. `Pandas` [6]. Una de las funciones con las que cuenta esta librería es la lectura de archivos con extensión `csv` y la manipulación de *data frames* facilitando el uso de tablas.
6. `Seaborn` [7]. Se utiliza para la creación del diagrama de caja y bigotes.
7. `Statmodels` [8]. Contiene varios modelos estadísticos y entre ellos esta el `ols` que sirve para ajustar formulas y entre los modelos a usar esta el `anova_lm` y `ols`. ←

Código 7: *Librerías utilizadas en la elaboración del código*

```
1 from statsmodels.formula.api import ols
2 import matplotlib.pyplot as plt
3 import statsmodels.api as sm
4 import networkx as nx
5 import seaborn as sns
6 import pandas as pd
7 import numpy as np
8 import time
```

Para guardar imagenes en formato ~~eps~~ se utiliza el comando que proporciona el fragmento de código 8.

Código 8: *Guardar imágenes en formato eps*

```
1 plt.savefig("Corrida" + str(contador) + ".eps")
```

Para evitar que los bordes de una imagen se recorten al momento de guardarla se le aplica el proceso que se muestra en el código 9.

Código 9: *Aumentar bordes para evitar cortes en el grafo*

```
1 plot_margin = 0.05
2
3 x0, x1, y0, y1 = plt.axis()
4
5 plt.axis((x0 - plot_margin,
6           x1 + plot_margin,
7           y0 - plot_margin,
8           y1 + plot_margin))
```

El objetivo de la experimentación es realizar una comparación entre los tiempos de ejecución tomando en cuenta varios factores para resolver un problema de máximo flujo, para eso, se utiliza una laptop con sistema operativo de 64 bits y un procesador ~~AMD A9-9410~~ ~~Radeon R5, 5 compute cores 2C+3G 2.90 GHz.~~

cinco núcleos

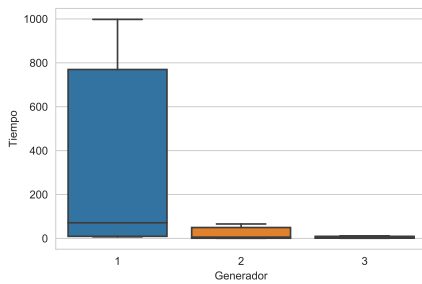
Para realizar la experimentación se consideran los siguientes parámetros:

- 3 ~~generadores~~ *grafos* (Grid, Complete y Circular ladder).
- 3 ordenes (27, 81 y 243).
- 10 ~~grafos~~ *grafos* distintos.
- 3 ~~implementaciones~~ *implementaciones* (Maximum flow, Minimum cut value y Maximum flow value).
- 5 parejas fuente-destino (aleatorias).

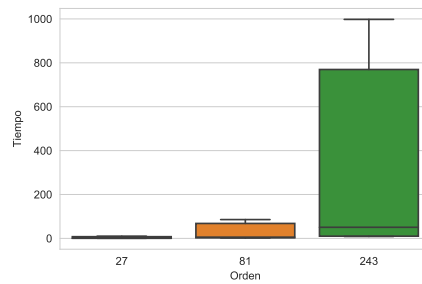
Se les asigna la variable aleatoria P a las capacidades de los arcos tal que $P \sim N(15, 5)$ y además para evitar tiempos nulos se consideran 100 repeticiones.

3. Experimentación

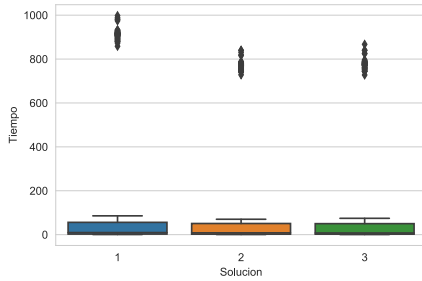
Una vez que se genera la matriz de datos con la caracterización de cada observación y su respectivo tiempo de ejecución se procede a la siguiente fase que es la experimentación estadística. En la figura 2 se observa como se comporta el tiempo de ejecución (en segundos) para los distintos grupos y niveles de cada grupo.



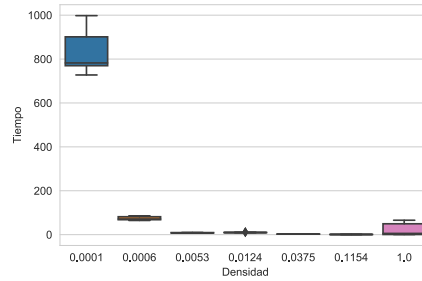
(a) Algoritmo generador



(b) Orden logarítmico



(c) Algoritmo de solución



(d) Densidad del grafo

Figura 2: Variación del tiempo de ejecución por nivel de los diferentes grupos.

En la figura 2 (a) se observa que el generador 1 es el que consume mayor tiempo y es un resultado que se esperaba puesto que este generador se aplica para crear grafos en 2 dimensiones, en (b) se hace énfasis a que entre mayor orden, mayor tiempo de ejecución, en (c) podría parecer que el efecto del algoritmo es nulo, es decir que no aportan variación en el tiempo de ejecución y en (d) se observa que a menor densidad mayor tiempo de ejecución aunque cuando la densidad es uno, también aumenta un poco el tiempo.

Posteriormente se efectúa un análisis de varianza (ANOVA) tipo dos para determinar los factores que tienen efecto sobre el tiempo, también es considerada la interacción entre los factores en el modelo para saber el tipo de efecto en la variable de respuesta.

Se considera el siguiente modelo matemático:

$$Y_{ijkl} = \mu_{...} + \alpha_i + \beta_j + \gamma_k + \delta_l + (\alpha\beta)_{ij} + (\alpha\gamma)_{ik} + (\alpha\delta)_{il} + (\beta\gamma)_{jk} + (\beta\delta)_{jl} + (\gamma\delta)_{kl} + \varepsilon_{ijkl}$$

Donde:

- $\mu_{...}, \alpha_i, \beta_j, \gamma_k, \delta_l, (\alpha\beta)_{ij}, (\alpha\gamma)_{ik}, (\alpha\delta)_{il}, (\beta\gamma)_{jk}, (\beta\delta)_{jl}, (\gamma\delta)_{kl}$ son parámetros que representan el efecto de cada nivel para cada factor simple y de interacción.
- $\varepsilon_{ijkl} \sim N(0, \sigma^2)$

y se realizan las siguientes pruebas estadísticas:

1. si cada factor tiene un efecto simple significativo en la variación del tiempo de ejecución,
2. si cada pareja de factores tiene un efecto de interacción significativo sobre el tiempo de ejecución.

En el cuadro 1 se muestran los resultados después de efectuar el ANOVA donde se destaca que dados los tiempos correspondientes se tiene que cada valor p (simple y de interacción) es menor a .05.

Cuadro 1: *Análisis de varianza del tiempo de ejecución.*

Fuente	Suma de cuadrados	Grados de libertad	Razón F	P(>F)
Generador	1.974622e+07	1	11894.0000	0.0000
Orden	3.012073e+07	1	18143.1582	0.0000
Solucion	7.021175e+04	1	42.2990	0.0000
Densidad	5.559616e+06	1	3348.8230	0.0000
Generador:Orden	1.292572e+07	1	7785.7826	0.0000
Generador:Solucion	8.767172e+04	1	52.6087	0.0000
Generador:Densidad	1.172786e+04	1	7.6425	0.0080
Orden:Solucion	9.736782e+04	1	58.6496	0.0000
Orden:Densidad	5.499527e+06	1	3312.6828	0.0000
Solucion:Densidad	2.271015e+04	1	13.6740	0.0002
Residual	2.222968e+06	1339	NA	NA

Del anova mostrado en el cuadro 1 podemos concluir que: se proporcionan valores p que son casi cero, por lo cual se puede decir que dada la elección de los parámetros mencionados en el ambiente computacional se obtiene una gran variación del tiempo de ejecución dado cualquier factor en el modelo y también cualquier interacción, es decir, si se cambia de nivel en algún factor por si solo o combinación de factores es muy seguro que haya grandes cambios en el tiempo de ejecución.

Referencias

- [1] NetworkX developers Versión 2.2. <https://networkx.github.io>.
- [2] Python Software Foundation Versión 3.7.2. <https://www.python.org/>.
- [3] The Matplotlib development team Versión 3.0.2. <https://matplotlib.org/>.
- [4] Sulce A. PythonHow. <https://pythonhow.com/measure-execution-time-python-code/>.
- [5] NumPy developers Versión 1.16.2. <https://networkx.github.io>.
- [6] Augspurger T. and the pandas core team Versión 0.24.2. <https://pandas.pydata.org/>.
- [7] Waskom M. Versión 0.9.0. Copyright 2012-2018. <https://seaborn.pydata.org/>.
- [8] Perktold J. and Seabold S. Versión 0.9.0. Copyright 2012-2018. <https://www.statsmodels.org/stable/index.html>.

Tarea 5

En esta práctica los errores fueron correspondientes a la falta de uso de notas de pie de página para especificar detalle, algunos errores de ortografía, descripción de graficas en el *caption* debajo de la figura.

Caracterización estructural de instancias

Mario Alberto Gutiérrez Carrales
1549273

30 de abril de 2019



Introducción

La presente práctica consiste en llevar a cabo cinco aplicaciones basadas en un caso de la vida real que sean consideradas como instancias del problema de máximo flujo de redes. Se analiza cada caso por separado realizando diferentes comparaciones de los elementos de la instancia para caracterizar a aquellos que la hacen más efectiva.

Los principales objetos de estudio que se consideran son los tiempos de ejecución y valores de flujo máximo ya que son quienes determinan la eficiencia de la instancia, los siguientes objetos con mayor importancia son los elementos del grafo, por ejemplo como los nodos y aristas así como la capacidad en las aristas y la cantidad que fluye entre ellas.

El lenguaje de programación que se utiliza es `python` [1] y una de las principales librerías que es necesaria para esta práctica es `NetworkX` [2] con funciones especiales para grafos, en especial para crear, acomodar y resolver problemas de optimización en redes utilizando distintos algoritmos.

1. Ambiente computacional

Los experimentos realizados en esta práctica se realizaron en una PC con un sistema operativo de 64 bits, procesador AMD A9-9410 Radeon R5, 2 núcleos + 3G con 2.90 GHz de velocidad y 12 GB de memoria RAM.


Para llevar a cabo la experimentación en **python** y para poder utilizar las funciones requeridas se necesitan las siguientes librerías:

1. **NetworkX**. Como se mencionó con anterioridad, esta librería es de gran ayuda para la manipulación de grafos.
2. **Matplotlib** [3]. Esta librería contiene funciones que sirven para visualizar gráficas y guardar imágenes en distintos formatos, en particular el *eps*.
3. **Time** [4]. Contiene la función que determina el tiempo (en segundos) que uno o varios procesos tardan, dicho tiempo se calcula con una diferencia entre el tiempo inicial y el final.
4. **Pandas** [5]. Una de las funciones con las que cuenta esta librería es la lectura de archivos con extensión *csv* y la manipulación de *data frames* facilitando el uso de tablas.
5. **Seaborn** [6]. Se utiliza para la creación del diagrama de caja y bigotes.

Dichas librerías se importan al entorno de **python** tal como se muestra en el código 1.

Código 1: *Cargar librerías en python*

```
1 import matplotlib.pyplot as plt
2 import networkx as nx
3 import seaborn as sns
4 import pandas as pd
5 import time
```


Para guardar imágenes en formato eps se utiliza el comando que proporciona el fragmento de código 2. 

Código 2: *Guardar imágenes en formato eps*

```
1 plt.savefig("CA_1.eps")
```

Para evitar que los bordes de una imagen se recorten al momento de guardarla se le aplica el proceso que se muestra en el código 3.

Código 3: *Aumentar bordes para evitar cortes en la imagen*

```
1 plot_margin = 0.05
2 x0, x1, y0, y1 = plt.axis()
3 plt.axis((x0 - plot_margin, x1 + plot_margin, y0 -
plot_margin, y1 + plot_margin))
```

2. Algoritmos para características de nodos

1. **degree centrality**. La centralidad del grado para un nodo v es la fracción de nodos a los que está conectado. Recibe como único parámetro un grafo.
2. **closeness centrality**. ~~Este algoritmo~~ ^{Este algoritmo} calcula la centralidad de cercanía de los vértices, tomando el recíproco del promedio de las distancias más cortas y normalizando. Recibe como parámetro principal un grafo y un parámetro opcional que es la normalización de los valores.
3. **clustering**. ~~Lo que mide este algoritmo es~~ ^{mide} el nivel de agrupamiento que existe entorno a un vértice, que se calcula como el número total de aristas que conectan a los vecinos más cercanos entre el número máximo de aristas posibles entre todos los vecinos más cercanos. Este algoritmo calcula el coeficiente de agrupamiento de los vértices y recibe como parámetro un grafo.
4. **load centrality**. ~~Es~~ ^{Es} la fracción de todas las rutas más cortas que pasan a través de ese nodo. Recibe como parámetro principal un grafo y un parámetro opcional que es la normalización de los valores.
5. **eccentricity**. ~~La excentricidad~~ ^{La excentricidad} de un nodo v es la distancia máxima de v a todos los demás nodos. Retorna la excentricidad de los nodos de un grafo y recibe como parámetros un grafo.
6. **pagerank**. ~~El algoritmo pagerank~~ ^{El algoritmo pagerank} calcula la probabilidad de que si uno navega las aristas del grafo aleatoriamente, vaya a dar a un nodo específico. Recibe un grafo como parámetro.

Footnote & ... ?

✗ Todos los valores retornados son guardados en una variable tipo diccionario en el que las claves son los nodos y los valores son los números que retorna el algoritmo. En el código 4 se muestra un ejemplo de como guardar dichos valores en un arreglo.

Código 4: Guardar los valores de los algoritmos en un arreglo

```
1 V1=[i for i in nx.degree_centrality(G).values() ]
2 V2=[i for i in nx.closeness_centrality(G).values() ]
3 V3=[i for i in nx.clustering(G).values() ]
4 V4=[i for i in nx.load_centrality(G).values() ]
5 V5=[i for i in nx.eccentricity(G).values() ]
6 V6=[i for i in nx.pagerank(G).values() ]
```

3. Metodología

Primero se elige una aplicación real, luego ciertos algoritmos para trabajar dicha aplicación y analizar diferentes variaciones. Para la construcción de cada instancia se toman en cuenta las siguientes características:

Aplicación. Un *sistema de tuberías* es un conducto que cumple la función de transportar agua u otros fluidos [7], en este caso supondremos que las aplicaciones son enfocadas en flujo de agua. Esta aplicación fue seleccionada debido a la gran importancia de los sistemas de tuberías en la industria y también por la eficiencia de un recurso vital como el agua, dependiendo de la situación dicho sistema debe cumplir con ciertas características físicas.

Algoritmo generador. *Barabasi_albert_graph* que toma dos parámetros, el primero es el número de nodos que tendrá el grafo y el segundo es el número de aristas que salen de cada nodo a otros. Se seleccionó este generador ya que gracias a su segundo parámetro se controla la cantidad de aristas presentes en el grafo y de esta manera aumentar su semejanza a una aplicación real en la que puede haber muchos nodos pero no tantas aristas.

Algoritmo de acomodo. *Kamada_kawai_layout* que toma como parámetro el grafo generado y retorna un diccionario con la la posición de cada nodo. Este algoritmo fue seleccionado gracias a que en la práctica dos de Gutiérrez [8] se realizó una comparación y debido a las características del grafo es el que mejor se adapta visualmente.

Implementación para el problema de máximo flujo. *Maximum_flow* que toma tres parámetros, el grafo que se resolvera, junto con el nodo inicial y el nodo final, esta función retorna dos objetos, un número que es el valor de máximo flujo de la instancia y también un diccionario que contiene los valores de flujo entre cada arco. A pesar de que este es un algoritmo que le toma mayor tiempo en resolver la instancia a comparación de otros el hecho de poder conocer el flujo de cada arista hace relevante esta implementación.

En el código 5 se muestra como se puede llevar a cabo el proceso de generación, acomodo y solución de instancia.

Código 5: Proceso de solución de instancia

```
1 #Generador
2 G=nx.barabasi_albert_graph(8,2)
3 #Algoritmo de acomodo
4 nx.draw_networkx(G,nx.kamada_kawai_layout(G),with_labels=False)
5 #Implementacion para el problema de maximo flujo
6 Max_Flujo, Flujo_Arcos = nx.maximum_flow(G, 0,3)
```

En la figura 1 se muestra el resultado del grafo generado y acomodado con los algoritmos indicados anteriormente.

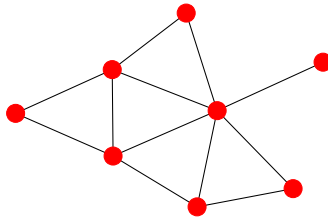


Figura 1: Grafo con algoritmo de acomodo

Posteriormente se lleva a cabo un análisis por instancia en el siguiente orden:

1. Visualización de instancia. Se muestran los nodos del grafo distinguiendo el inicio (rosa), final (rojo) y los intermedios (aqua). Además el grosor de la arista representa la capacidad en relación a las demás aristas.
2. Visualización de instancia con solución. Se muestra el grafo anterior añadiendo el flujo que corresponde a cada arista (celeste) preservando la idea de que el grosor del flujo indica una relación con los demás flujos, también se añade la variación de tamaño de los nodos de la instancia utilizando los algoritmos para características de nodos, esto con el fin de determinar cuales son los más relevantes (a mayor radio, mayor relevancia).
3. Visualización de instancia con solución y caracterización. Se toma el mismo gráfico anterior con la sutileza de que se le añade color a los nodos intermedios que cumplen un papel importante, es decir, que si se

varía el nodo inicial o final para dicha instancia podrían ser un buen nodo inicial o final, gracias a que se mejora la función objetivo. Los nodos que son buen inicio pero no final (**azul** y **verde limón**), aquellos que son mejor nodo final pero no inicial (**morado** y **verde limón**) y por último los nodos que son tanto mejor inicio como mejor final (**amarillo**).

4. Discusión de las características de nodos y parejas que mejoran el flujo a la instancia.
5. Comparación del tiempo de ejecución (TE) variando el nodo inicial.
6. Comparación del tiempo de ejecución (TE) variando el nodo final.
7. Comparación tiempo de ejecución (TE) variando un nodo intermedio eliminado.
8. Discusión del mejor nodo posible, tiempos de ejecución y posibles nodos a eliminar de la instancia.

Para el caso de las visualizaciones de las instancias, solamente en la grafica 1 se muestra el número de nodo.

Para los casos 5, 6 y 7 se grafican los diagramas de caja y bigote identificandp los casos en los que la función objetivo mejora (**rojo**) y los que permanecen igual o disminuyen (**azul**).

Después de analizar cada instancia, se realiza una comparación global mediante un diagrama de caja para cada una de las características de los nodos correspondientes a las diferentes instancias.

Instancia 1

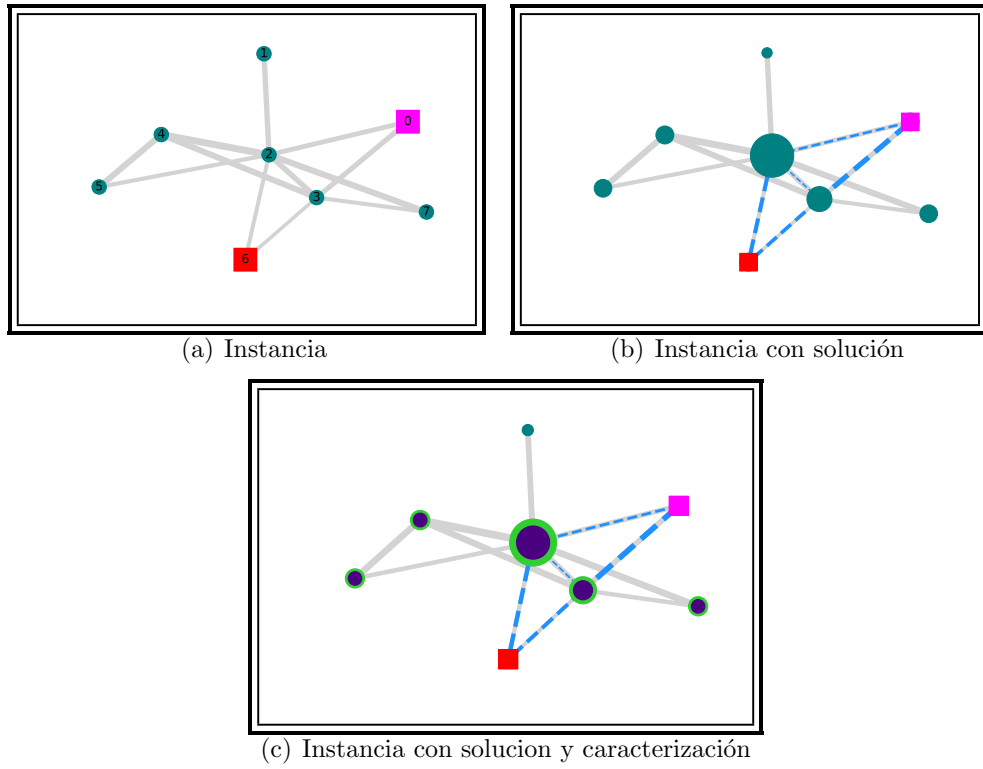
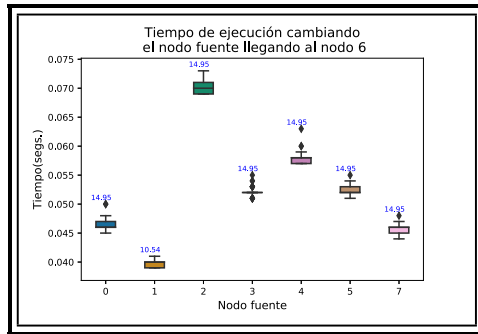


Figura 2: Visualización de la instancia 1.

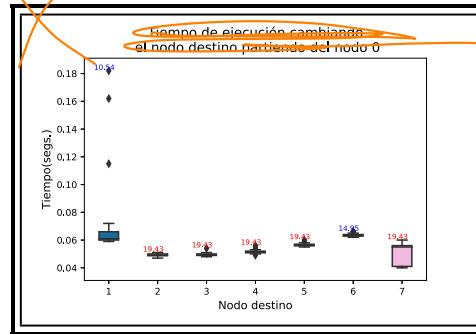
En la figura 2 se observa que del total de las aristas solo se usan 5, representando el 41.6 %.

También se muestra que los nodos 2 y 3 tiene en general mejores características que el resto y que para esta instancia los nodos 2, 3, 4, 5 y 7 son mejores nodos finales (destino) que el principal, obteniendo un aumento en la función objetivo.

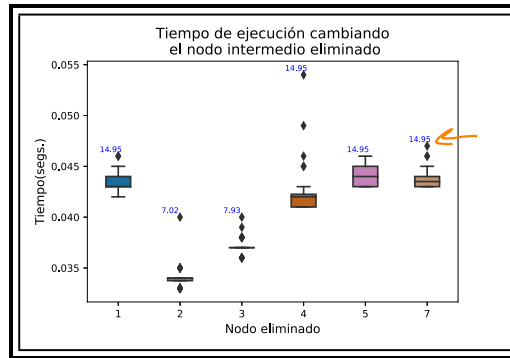
Por otro lado, si el problema se resuelve variando el nodo inicial no se obtiene mejora la función objetivo.



(a) Comparación de TE variando la fuente



(b) Comparación de TE variando el destino



(c) Comparación de TE variando eliminado

Figura 3: Comparación del tiempo de ejecución para la instancia 1.

En la figura 3 se observa que solamente empeora la función objetivo cuando se cambia el nodo inicial al 1, todos los demás son iguales al inicial, y los mejores en cuanto a tiempo de ejecución son el nodo 0 y el 7.

Como se mencionó en el apartado de visualización, cuando se varía el nodo final se obtienen varias mejoras, y de esas mejoras los que presentan mejores tiempos de ejecución son el nodo 2 y 7, sobretodo el 2.

Si se elimina un nodo intermedio, no se obtienen mejoras en la función objetivo, así que no conviene eliminar ningún nodo.

Instancia 2

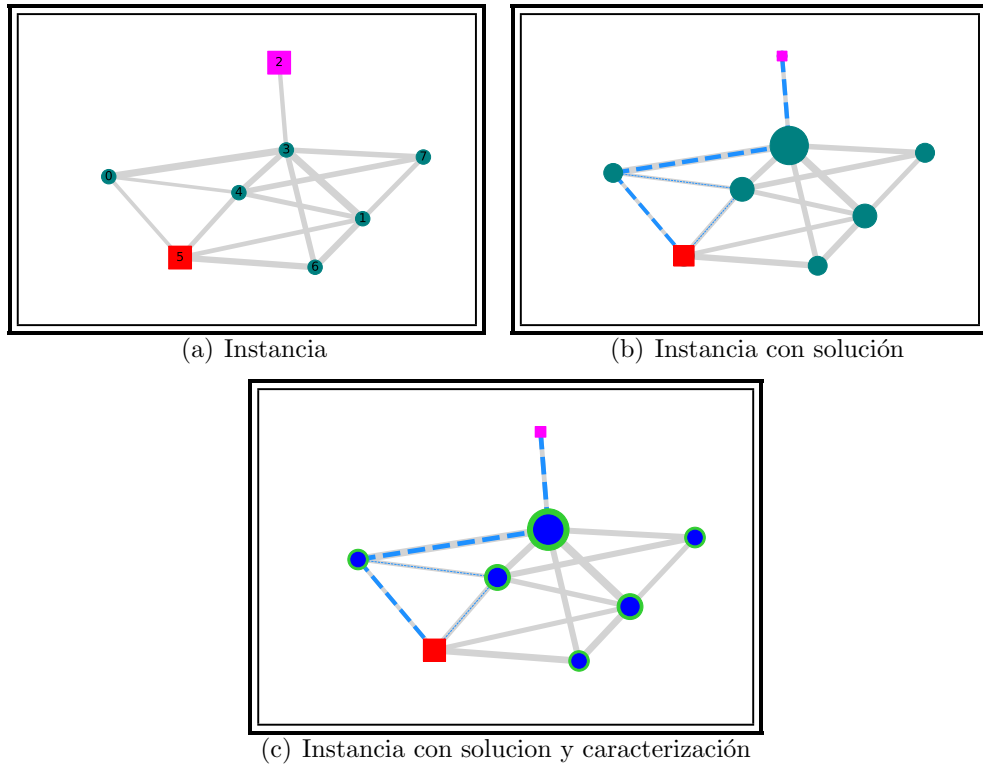


Figura 4: Visualización de la instancia 2.

En la figura 4 se observa que del total de las aristas solo se usan 5, representando el 33.3 %.

También se muestra que los nodos 1, 3 y 4 tiene en general mejores características que el resto y que para esta instancia todos los nodos intermedios son mejores nodos iniciales (fuente) que el principal, obteniendo un aumento en la función objetivo.

Por otro lado, si el problema se resuelve variando el nodo final no se obtiene mejora la función objetivo.

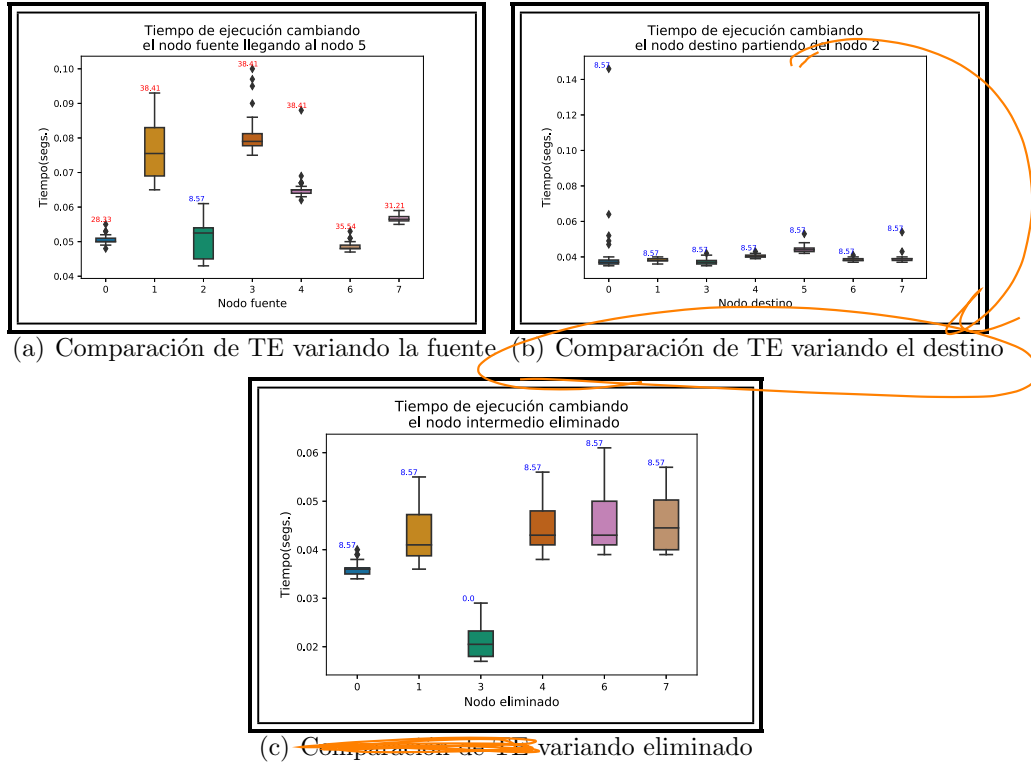


Figura 5: Comparación del tiempo de ejecución para la instancia 2.

En la figura 5 se observa que cuando se varía el nodo inicial se obtienen solamente mejoras, y de esas mejoras los nodos que presentan mejor función objetivo son el 1, 3 y 4, de los cuales el que presenta mejor tiempo de ejecución es el 3.

También se observa que fijando el nodo inicial y cambiando el nodo destino siempre se obtiene el mismo valor en la función objetivo, gracias a eso, y a la comparación de tiempos de ejecución, se podría considerar que los mejores nodos el 1 y el 7.

Si se elimina un nodo intermedio, no se obtienen mejoras en la función objetivo, así que no conviene eliminar ningún nodo, de hecho, si se elimina el nodo 3, el problema es infactible, por eso la importancia de este nodo.

Instancia 3

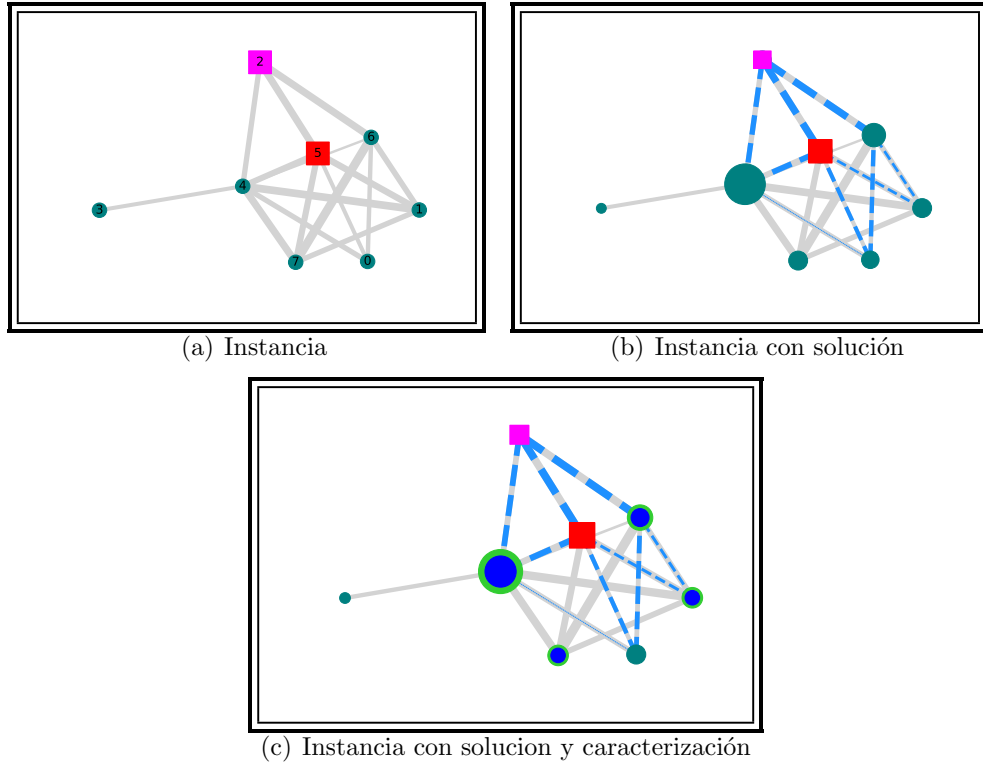


Figura 6: Visualización de la instancia 3.

En la figura 6 se observa que del total de las aristas solo se usan 9, representando el 60 %.

También se muestra que los nodos 4 y 5 tiene en general mejores características que el resto y que para esta instancia los nodos 1, 4, 6 y 7 son mejores nodos iniciales (fuente) que el principal, obteniendo un aumento en la función objetivo.

Por otro lado, si el problema se resuelve variando el nodo final no se obtiene mejora la función objetivo.

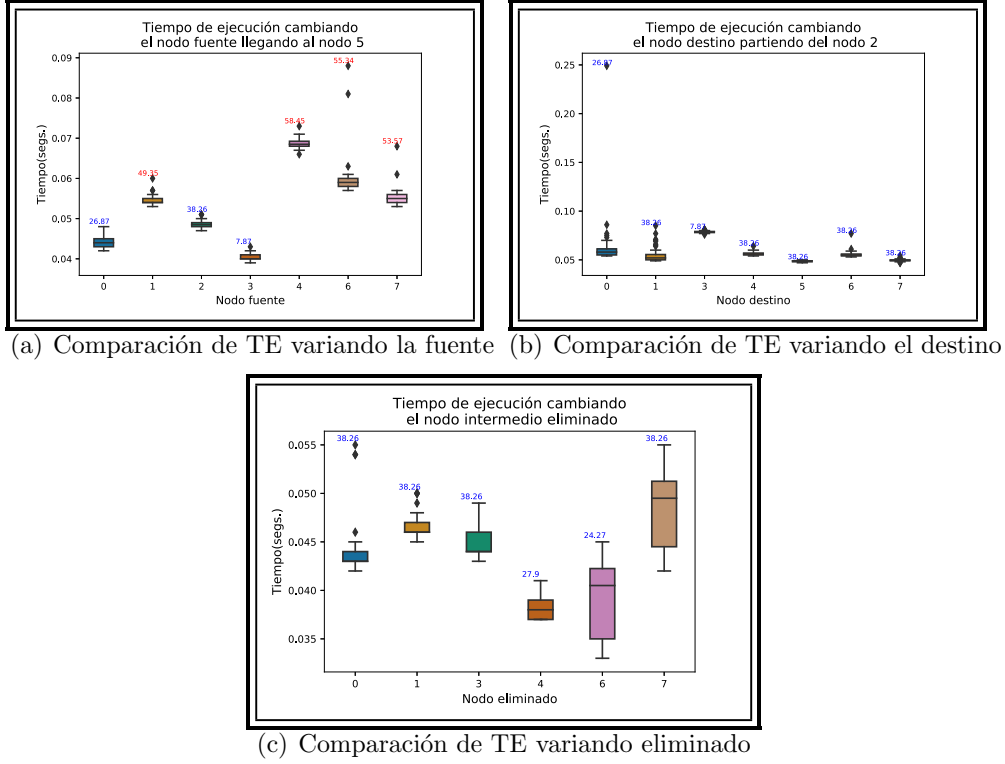


Figura 7: Comparación del tiempo de ejecución para la instancia 3.

En la figura 7 se observa que si se resuelve la instancia cambiando el nodo inicial, se obtienen diferentes mejoras, los nodos para los cuales se presenta eso son el 1, 4, 6 y 7 y de estos el que obtiene mayor función objetivo es el 4, pero también es el que más tiempo le toma en resolver la instancia.

Como se mencionó en el apartado de visualización, cuando se varía el nodo final no se obtiene ninguna mejora, pero si se obtienen valores que empatan en función objetivo con la instancia original, los nodos para los que ocurre esto, son el 1, 4, 6 y 7, de los cuales los que se ejecutan en menor tiempo son el 7 y la instancia original con el nodo destino 5.

Si se elimina un nodo intermedio, no se obtienen mejoras en la función objetivo, pero si hay quienes se pueden eliminar y se conserva el valor del flujo óptimo, en este caso, se puede eliminar el nodo 3 por estar aislado.

Instancia 4

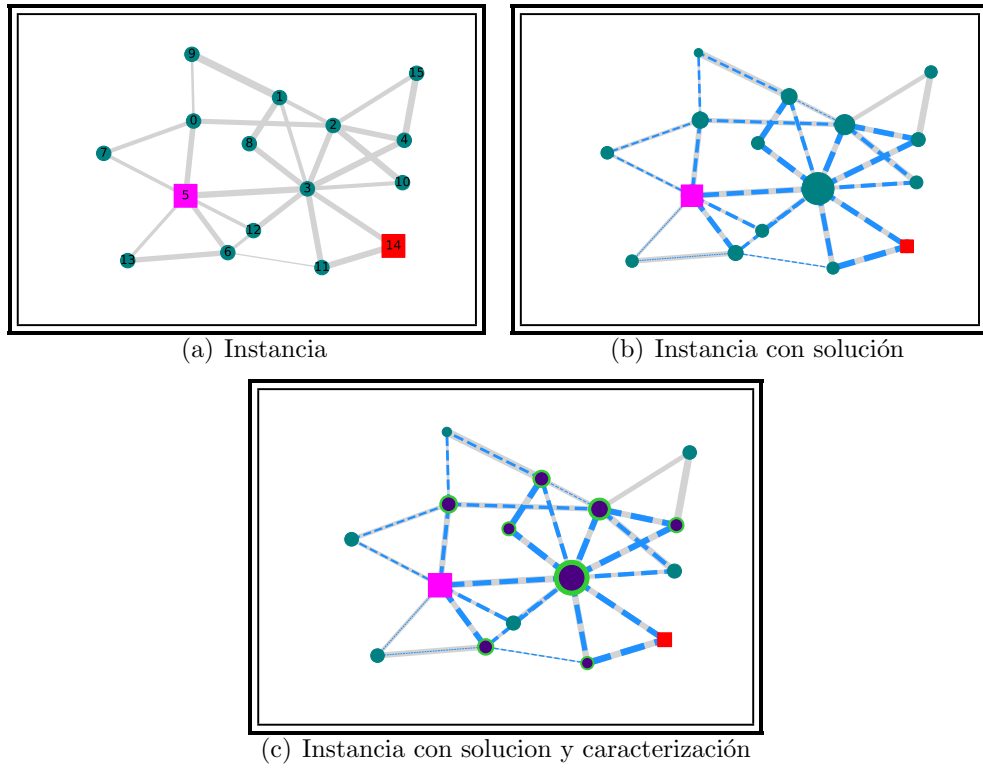


Figura 8: Visualización de la instancia 4.

En la figura 8 se observa que del total de las aristas se usan 26, representando el 92.85 %.

También se muestra que los nodos 3 y 5 tienen en general mejores características que el resto y que para esta instancia los nodos 0, 1, 2, 3, 4, 6, 8 y 11 son mejores nodos finales (destino) que el principal, obteniendo un aumento en la función objetivo.

Por otro lado, si el problema se resuelve variando el nodo inicial no se obtiene mejora la función objetivo.

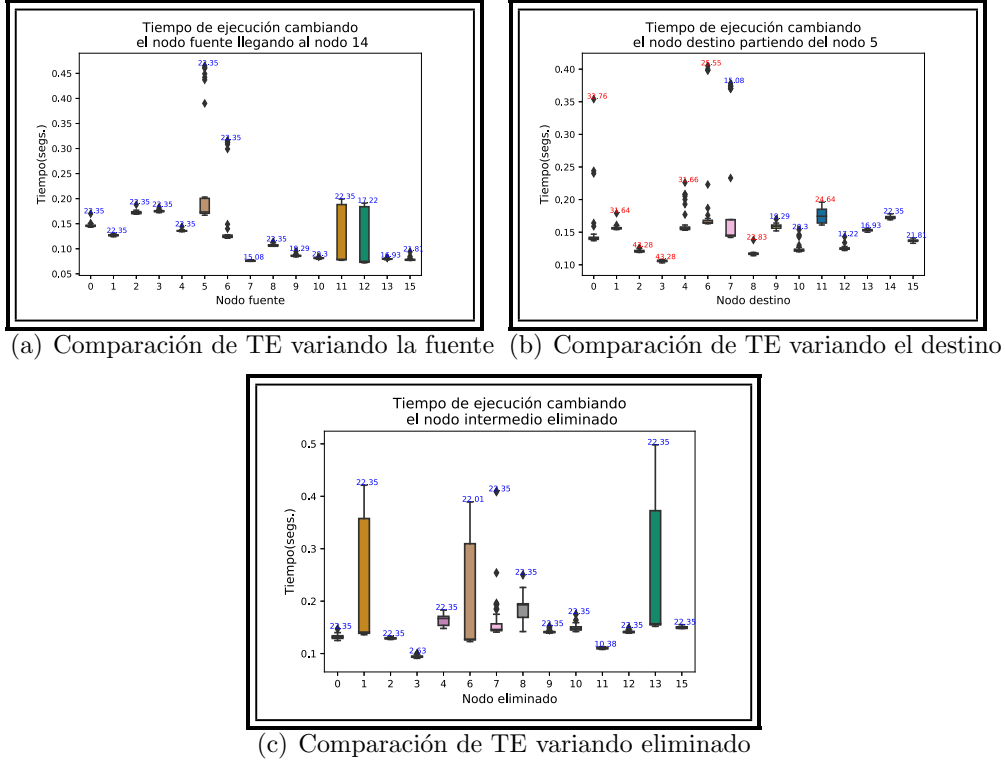


Figura 9: Comparación del tiempo de ejecución para la instancia 4.

En la figura 9 se observa que no hay mejoras en la función objetivo variando el nodo inicial y los que la empeoran son los nodos 7, 9, 10, 12, 13 y 15, el resto conserva el mismo flujo que el nodo inicial original, de los cuales el nodo 8 es el que presenta menor tiempo de ejecución al resolver la instancia.

Como se mencionó en el apartado de visualización, cuando se varía el nodo final se obtienen varias mejoras, y de esas mejoras los que presentan mayor función objetivo son los nodos 2 y 3, siendo el 2 el que presenta menor tiempo de ejecución.

Si se elimina un nodo intermedio, no se obtienen mejoras en la función objetivo, pero para algunos se conserva el flujo de la instancia original, así que se pudiera eliminar alguno de ellos, los mejores candidatos son 1, 9, 12 y 15 ya que son los que hacen que la distancia se resuelva más rápido.

Instancia 5

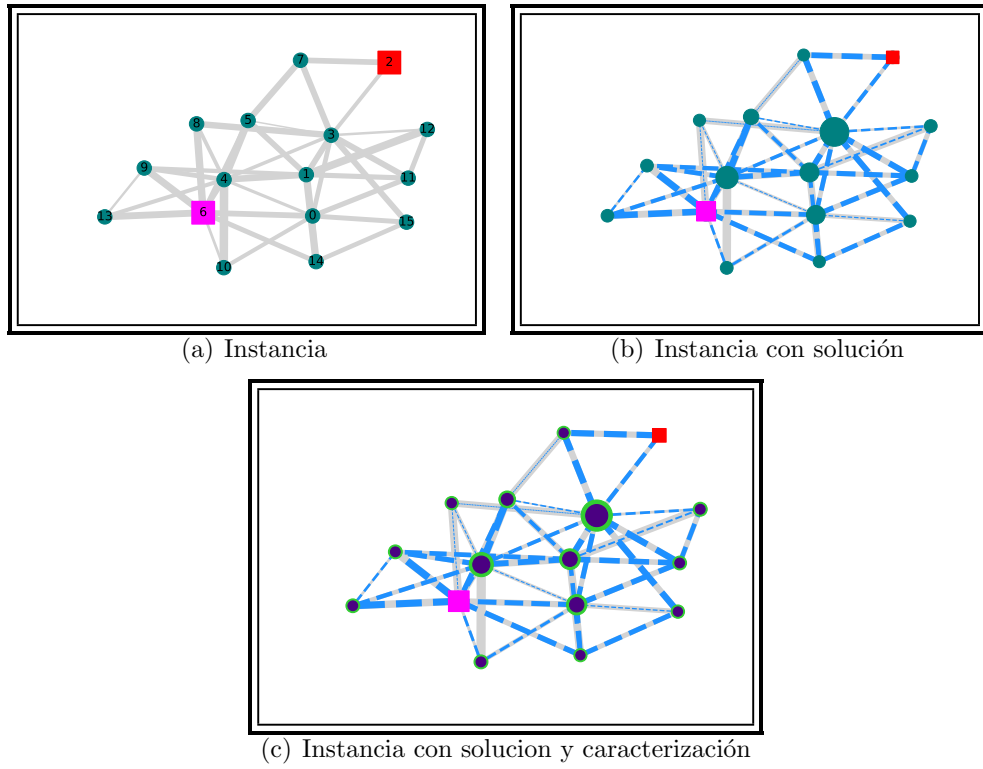
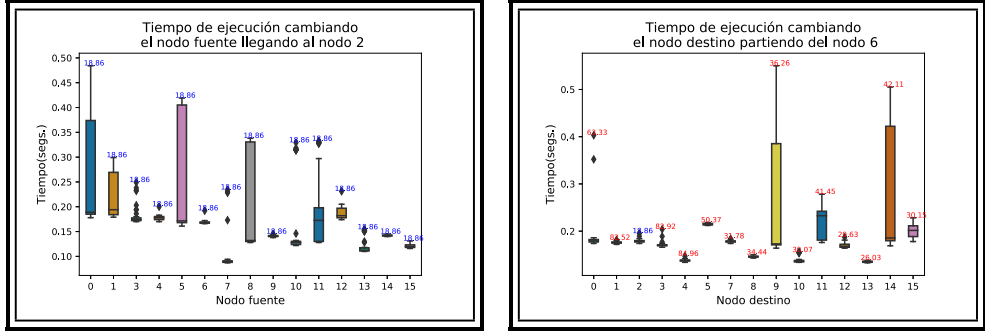


Figura 10: *Visualización de la instancia 5.*

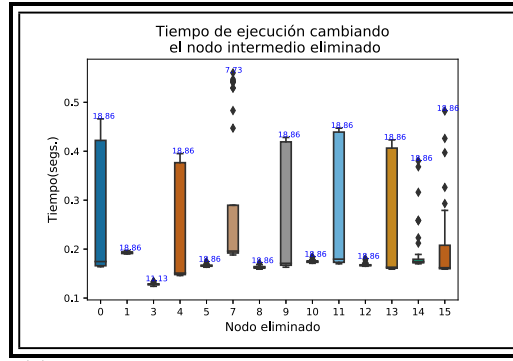
En la figura 10 se observa que del total de las aristas se usan 28, representando el 96.55 %.

También se muestra que los nodos 3 y 4 tiene en general mejores características que el resto y que para esta instancia todos los nodos intermedios son mejores nodos finales (destino) que el principal, obteniendo un aumento en la función objetivo.

Por otro lado, si el problema se resuelve variando el nodo inicial no se obtiene mejora la función objetivo.



(a) Comparación de TE variando la fuente (b) Comparación de TE variando el destino



(c) Comparación de TE variando eliminado

Figura 11: Comparación del tiempo de ejecución para la instancia 5.

En la figura 11 se observa que no importa que nodo se tome como nodo fuente siempre se obtiene el mismo flujo y de estos el que resuelve la instancia más rápido es el nodo 7, presentando algunos casos atípicos con mayor tiempo.

Por otro lado, como se mencionó en el apartado de visualización cuando se varía el nodo final todos brindan mejor función objetivo, en especial los nodos 1, 3 y 4, siendo este ultimo el de mayor valor y el de menor tiempo de ejecución.

Si se elimina un nodo intermedio, no se obtienen mejoras en la función objetivo, pero para algunos se conserva el flujo de la instancia original, así que se pudiera eliminar alguno de ellos, los mejores candidatos son 5, 8, 10 y 12 ya que son los que hacen que la distancia se resuelva más rápido.

Comparación de instancias

En este ultimo apartado, se comparan las características de los nodos para cada grafo, esperando poder determinar que característica pudiera parecer importante para un flujo más grande.

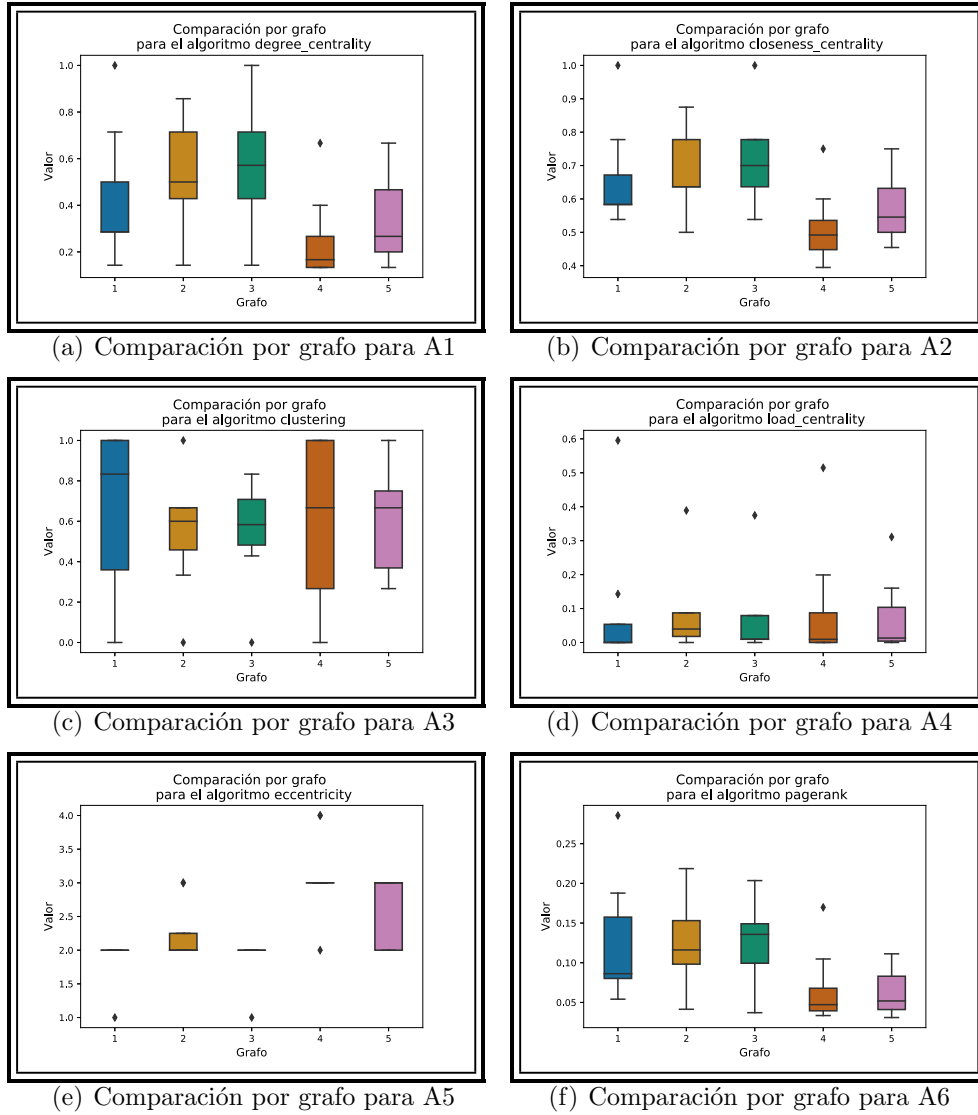


Figura 12: Comparación de los valores por grafo para cada característica.

Recordando los valores objetivos 14.95, 8.57, 38.26, 23.25 y 18.86, así como que el número de nodos de las primeras 3 instancias son de 8 nodos y el resto de 16, para las instancias 1, 2, 3, 4 y 5, respectivamente.

De acuerdo a los resultados de las comparaciones de las características se obtienen las siguientes conclusiones generales:

1. Dado que la instancia 3 fue la que mejor flujo obtuvo y la 4 fue la que le siguió pero obtuvieron buen y mal valor para el `degree centrality` respectivamente, no pareciera que este algoritmo influencie en la función objetivo.
2. Si se obtiene bajo valor de `closeness centrality` obtiene mayor flujo, mientras que un alto valor, provoca menor flujo en el grafo.
3. En el algoritmo `clustering` hay mucha interacción entre los grafos y las funciones objetivos, dicho eso, no podemos decir que haya una relación lineal entre el valor de este algoritmo presente en el grafo y la función objetivo.
4. Las instancias 2 y 3 son la de menor y mayor objetivo respectivamente, pero para este algoritmo `load centrality` son los que mayor valor tienen, así que, pareciera que no habría una relación clara en base a esta característica.
5. En este caso, la instancia 4 obtuvo muy buen valor de `eccentricity`, pero la instancia 3 muy malo, así que parece que habría contraste en los resultados de funciones objetivos, pareciendo que no hubiera relación.
6. Para este algoritmo `pagerank` parece haber una tendencia entre que a mayor valor, menor flujo y para menor valor, mayor flujo.

Estas conclusiones se obtuvieron a partir de las instancias realizadas, si se desea mayor precisión, se puede aumentar el número de instancias y de variaciones en ellas.

Referencias

- [1] Python Software Foundation Versión 3.7.2. <https://www.python.org/>.
- [2] NetworkX developers Versión 2.2. <https://networkx.github.io>.
- [3] The Matplotlib development team Versión 3.0.2. <https://matplotlib.org/>.
- [4] Sulce A. PythonHow. <https://pythonhow.com/measure-execution-time-python-code/>.
- [5] Augspurger T. and the pandas core team Versión 0.24.2. <https://pandas.pydata.org/>.
- [6] Waskom M. Versión 0.9.0. Copyright 2012-2018. <https://seaborn.pydata.org/>.
- [7] Francois J. Mecánica de fluidos. Escuela de Ingeniería Mecánica.
- [8] Gutiérrez M. Repositorio optimización flujo en redes. https://github.com/MarioGtz14/Flujo_Redes_Mario.

Tarea 6

Esta práctica no se había revisado, se adjunta para su revisión.

El problema de eliminación de béisbol

Mario Alberto Gutiérrez Carrales
1549273

20 de mayo de 2019

1. Introducción

El problema de máximo flujo es un problema muy relevante dentro del campo de la optimización en redes ya que es usado como herramienta para resolver otros problemas que son más elaborados, tales como el problema de emparejamiento máximo, el problema de corte mínimo, y problemas de flujo en procesos en general.

El objetivo que se busca cumplir es modelar mediante una red una instancia del problema de eliminación de béisbol como una para el problema de flujo máximo y posteriormente resolverla utilizando una implementación para el problema de máximo flujo que la librería NetworkX ofrece.

2. Ambiente computacional

Las implementaciones computacionales se realizan en **python** [1] utilizando librerías como la ya mencionada **NetworkX** [2] que sirve para la manipulación de grafos, así como algoritmos de solución para problemas de optimización en redes y **Matplotlib** [3] que es empleada para la visualización de gráficos.

Los experimentos realizados en esta práctica se realizaron en una PC con un sistema operativo de 64 bits, procesador AMD A9-9410 Radeon R5, 2 núcleos + 3G con 2.90 GHz de velocidad y 12 GB de memoria RAM.

3. El problema de eliminación de béisbol

El problema de eliminación de béisbol (que también aplica para hockey y basquetbol) se popularizó en 1960 por Alan Hoffman [4, 5, 6]

El problema de eliminación de béisbol se define de la siguiente manera:

Dada la clasificación en una división deportiva en algún momento de la temporada, determine si un equipo ha sido eliminado matemáticamente y no tiene oportunidad de ganar su división. (No se permiten empates en los encuentros)

Supóngase que se tiene la siguiente instancia del problema de eliminación de béisbol

Equipo	PG	PR	1	2	3	4
1	33	8	-	1	6	1
2	29	4	1	-	0	3
3	28	7	6	0	-	1
4	27	5	1	3	1	-

Es decir, 4 equipos se están disputando el campeonato y en la tabla se tienen los partidos que han ganado hasta el momento (PG), los partidos restantes que le restan por jugar a cada equipo (PR) y la matriz de juegos pendientes entre los equipos.

Se desea saber si el equipo 2 está eliminado o no.

Primero se puede observar que el número máximo de partidos ganados que puede conseguir el equipo 2 es $PG_n(2) = 29 + 4 = 33$ (si gana todos sus juegos).

Si gana todos sus juegos, no será eliminado solamente si:

- 1 no gana más de $C(1) = PG_n(2) - PG(1) = 33 - 33 = 0$ en sus juegos restantes.
- 3 no gana más de $C(3) = PG_n(2) - PG(3) = 33 - 28 = 5$ en sus juegos restantes.
- 4 no gana más de $C(4) = PG_n(2) - PG(4) = 33 - 27 = 6$ en sus juegos restantes.

Sea P el conjunto de equipos sin considerar el equipo 2 y Q el conjunto de todos los posibles juegos entre los equipos en P .

$$P = \{1, 3, 4\} \quad Q = \{(1, 3), (1, 4), (3, 4)\}$$

Para crear la red se toma en cuenta las siguientes consideraciones:

- Crear un nodo fuente O (Todos los juegos se originan de aquí).
- Crear un nodo por cada pareja en Q . Para cada nodo (i, j) creado, agregar un arco de O a (i, j) , la capacidad del arco es el número de juegos que restan por jugar entre i y j .
- Crear un nodo por cada equipo en P . Para cada nodo $(i, j) \in Q$ agregar arcos a i y a j cumpliendo la relación $\text{capacidad}((i, j) \rightarrow i) = \text{capacidad}((i, j) \rightarrow j) = \text{capacidad}(O \rightarrow (i, j))$.
- Crear un nodo sumidero T (Las victorias de los equipos se registran aquí). Agregar arcos desde todo nodo $i \in P$ a T , la capacidad de arco es $C(i)$.

Encontrar el flujo máximo desde O a T en la red resultante.

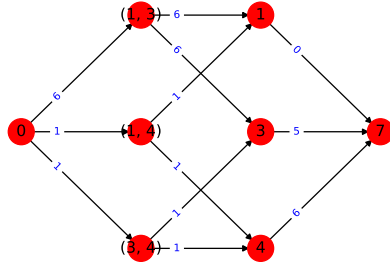
Si el valor de flujo máximo es igual al número total de juegos restantes entre los equipos en P entonces el equipo 2 aún tiene oportunidades de terminar como primero, sino ya está eliminado.

En este caso el número de juegos pendientes entre los equipos en P es 8, así que si el valor de flujo es menor a 8, entonces el equipo 2 está eliminado.

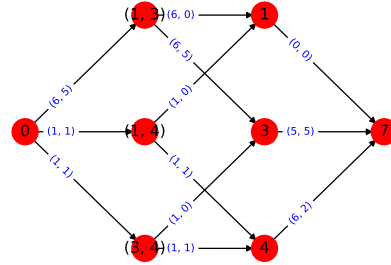
En la figura 1 se muestran las redes que representan la instancia al problema de eliminación de béisbol y la solución a dicha instancia respectivamente.

En la red de instancia se muestran las etiquetas de las capacidades de los arcos, mientras que en la red de solución se muestra la pareja ordenada (flujo permitido, flujo usado) para cada arco.

Posteriormente se encuentran los fragmentos de código 1, 2, 3, 4, 5 y 6 en donde se muestra como se configuran los parámetros de la instancia, los nodos y las conexiones entre ellos, la grafica de la instancia, guardar una imagen en formato *eps*, crear parejas y finalmente graficar la solución respectivamente.



(a) Instancia



(b) Solución

Figura 1: Instancia y solución del problema de eliminación de béisbol.

Código 1: Parámetros de la instancia

```

1 Equipos = [i+1 for i in range(4)]
2 equipo = 2
3 cap=[[6,1,1],[0,5,6]]
4
5 P=[i for i in Equipos if i!=equipo]
6 Q=[(i,j) for i in P for j in P if i<j]

```

Código 2: Crear conexiones

```

1 G = nx.DiGraph()
2 O=0
3 G.add_nodes_from([O])
4 G.add_nodes_from(Q)
5 for i in range(len(Q)):
6     G.add_edge(O,Q[i],capacity=cap[0][i])
7 G.add_nodes_from(P)
8 for i in range(len(Q)):
9     for j in range(2):
10        G.add_edge(Q[i],Q[i][j],capacity=cap[0][i])
11 T=len(P)+len(Q)+1
12 G.add_nodes_from([T])
13 for j in range(len(P)):
14     G.add_edge(P[j],T,capacity=cap[1][j])

```

Código 3: *Graficar la instancia*

```
1 nx.draw_networkx_nodes(G,pos,node_shape='o',node_size=400)
2 nx.draw_networkx_edges(G,pos)
3 nx.draw_networkx_labels(G,pos)
4 nx.draw_networkx_edge_labels(G,pos,label_pos=0.7,font_size=8,edge_labels=nx
    .get_edge_attributes(G,'capacity'),font_color='b')
```

Código 4: *Guardar imagen en formato eps*

```
1 plt.axis('off')
2 plt.savefig('Instancia.eps')
```

Código 5: *Crear parejas de flujo permitido y usado*

```
1 for (i,j) in nx.get_edge_attributes(G,'capacity').keys():
2     a = Capacidad[(i,j)] #Lo que puede pasar
3     b = arcos[i][j]      #Lo que pasa
4     dic_solcap[(i,j)]=(a,b)
```

Código 6: *Graficar la solución*

```
1 nx.draw_networkx_nodes(G,pos,node_shape='o',node_size=400)
2 nx.draw_networkx_edges(G,pos)
3 nx.draw_networkx_labels(G,pos)
4 nx.draw_networkx_edge_labels(G,pos,label_pos=0.7,font_size=8,edge_labels=
    dic_solcap,font_color='b')
```

En base a la solución mostrada en la subfigura b) de la figura 1 se puede identificar que el valor de flujo máximo es 7, el cual es menor que el número de juegos pendientes entre los equipos en P , lo que quiere decir que el equipo 2, está matemáticamente eliminado.

Referencias

- [1] Python Software Foundation Versión 3.7.2. <https://www.python.org/>.
- [2] NetworkX developers Versión 2.2. <https://networkx.github.io>.
- [3] The Matplotlib development team Versión 3.0.2. <https://matplotlib.org/>.
- [4] Boxley L. *An application of Maximum flow: The baseball elimination problem*. <https://slideplayer.com/slide/4016790/>.
- [5] Wayne K. *Baseball elimination*. Princeton University.
- [6] Allen P. *Introduction to algorithms*. University of Washington.