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 librerias 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 margenes 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*

```
plot_margin = 0.05

x0, x1, y0, y1 = plt.axis()

plt.axis((x0 - plot_margin,
          x1 + plot_margin,
          y0 - plot_margin,
          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.

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 asi 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. all 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 solucion 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')
```

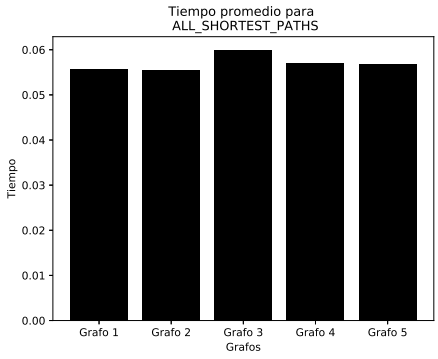
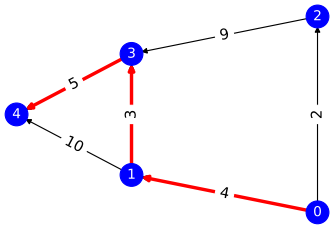
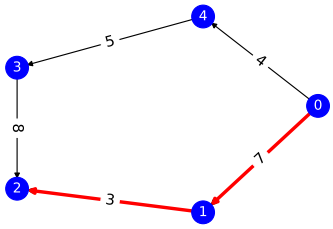


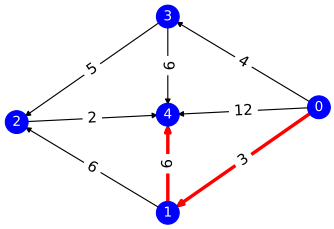
Figura 1: *Histograma de los tiempos promedio por grafo*



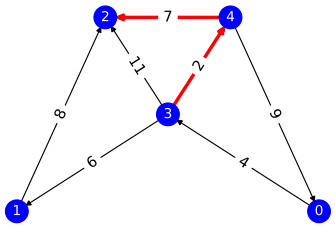
(a) Grafo 1



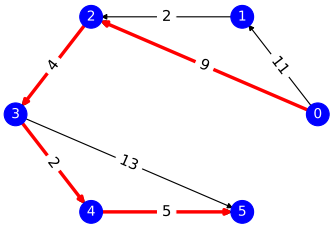
(b) Grafo 2



(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)
```

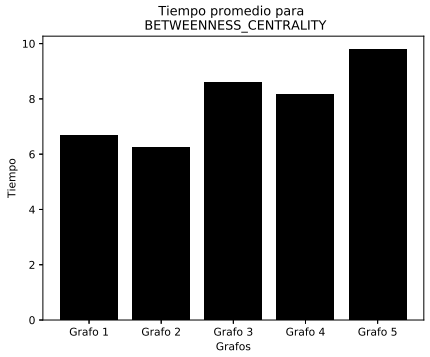
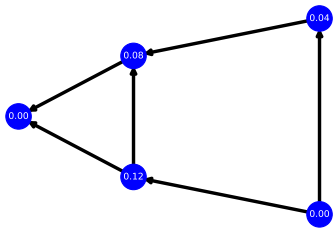
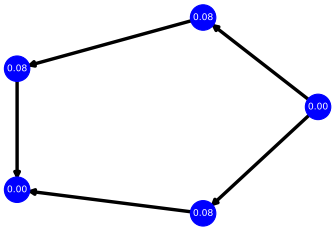


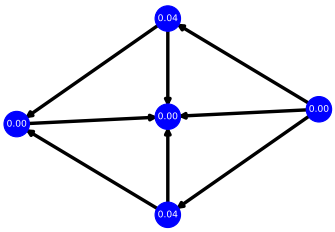
Figura 3: *Histograma de los tiempos promedio por grafo*



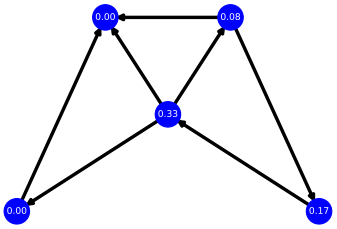
(a) Grafo 1



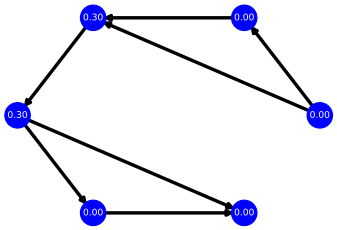
(b) Grafo 2



(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')
```

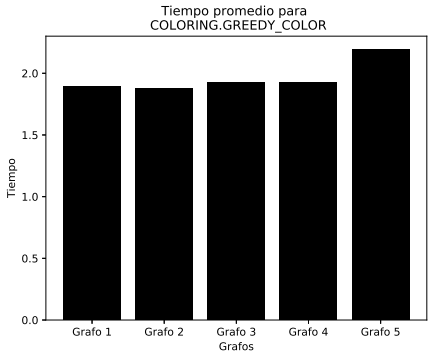
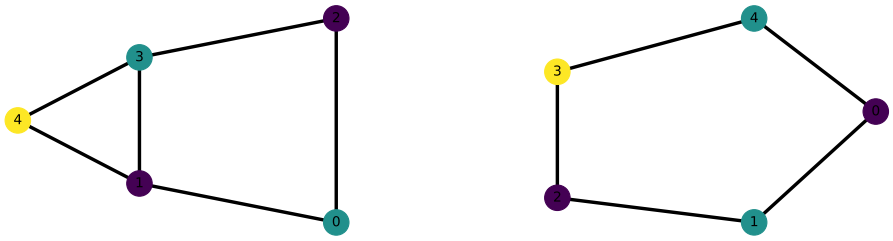
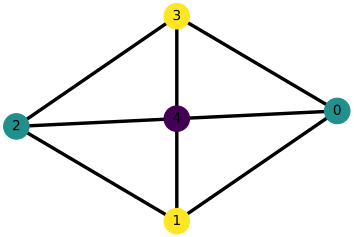


Figura 5: *Histograma de los tiempos promedio por grafo*

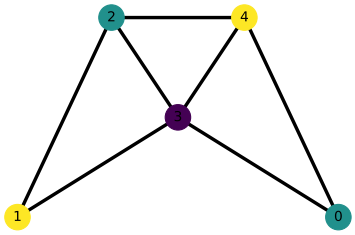


(a) Grafo 1

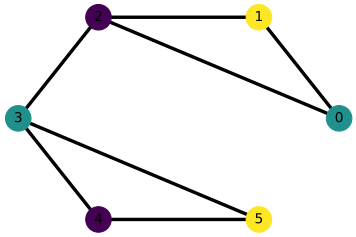
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 6: *Solución al coloreo de grafos.*

4. maximum flow

Este algoritmo resuelve el problema de máximo flujo en una red, esta función toma como parámentros 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')
```

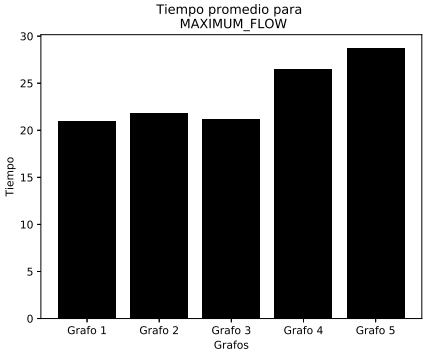
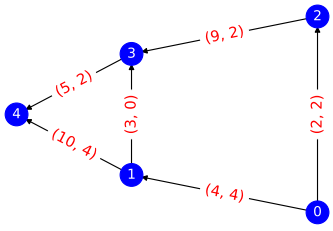
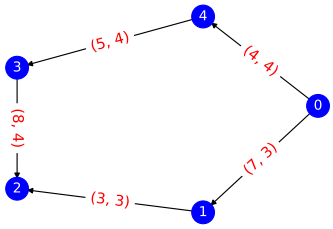


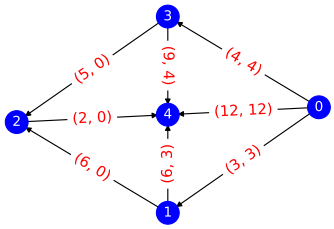
Figura 7: *Histograma de los tiempos promedio por grafo*



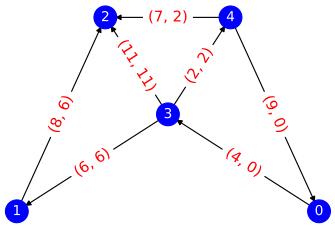
(a) Grafo 1



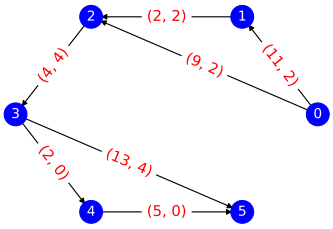
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 8: *Solución del máximo flujo en una red.*

5. minimum 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
Si	Si	Si	Si*

*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')
```

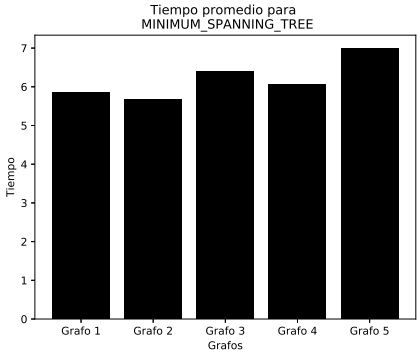
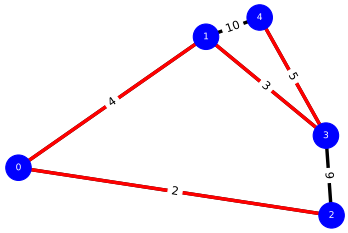
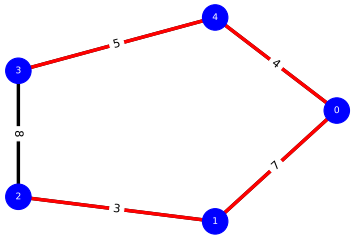


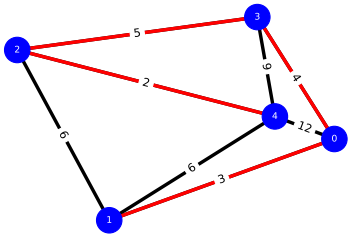
Figura 9: *Histograma de los tiempos promedio por grafo*



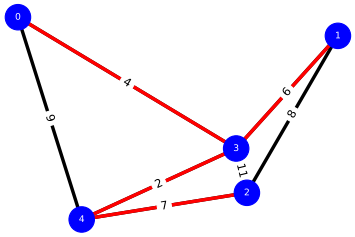
(a) Grafo 1



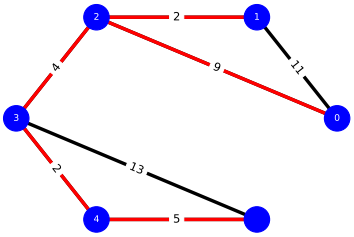
(b) Grafo 2



(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.

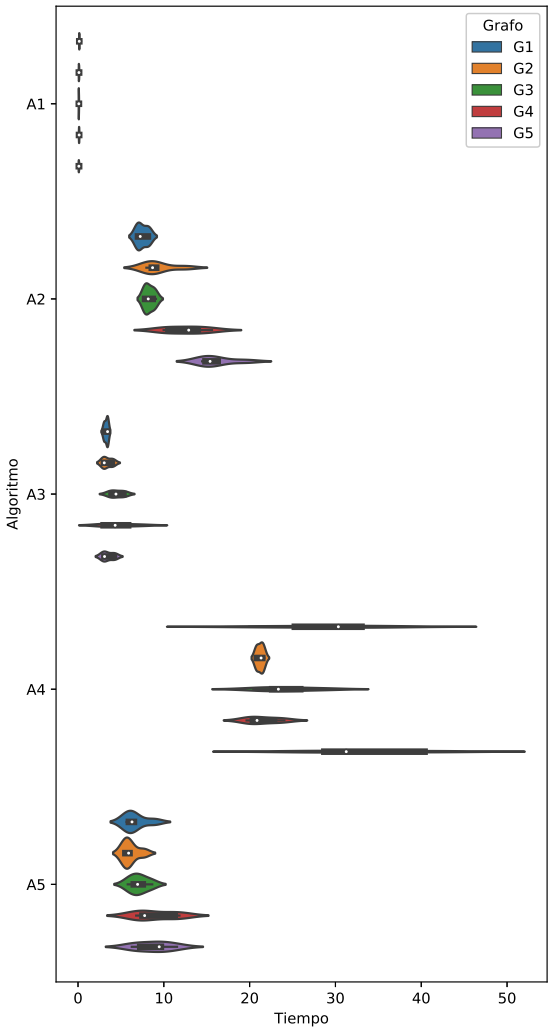


Figura 11: *Grafica de violín para los algoritmos y grafos*

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.