

FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA (FIME)

MAESTRÍA EN CIENCIAS EN INGENIERÍA DE SISTEMAS

PORTAFOLIO DE EVIDENCIA

Optimización de Flujo de Redes

ALUMNO:
Evely Gutiérrez Noda

PROFESOR:
Dr. Elisa Schaeffer

Enero - Junio 2018

1. Introducción

El reporte muestra un resumen por tareas realizadas a lo largo del semestre en la asignatura optimización de Flujo de Redes. También se desarrolla una retroalimentación por parte del estudiante luego de la revisión de cada tarea por parte del profesor. la revisión de la tarea se presenta luego de cada retroalimentación.

2. Discusión de la Tarea 1

La **Tarea 1** fue basada en la construcción de grafos utilizando lenguaje **Python** para la programación cuando se desea crear un grafo, además se utilizó la herramienta **Gnuplot**, la cual se vinculó con **Python** para representar los grafos programados.

Se crearon grafos de distintos tamaños, variando la cantidad de nodos y creando conexiones entre estos nodos con aristas de forma aleatoria. Estos conjuntos de nodos y aristas creados se guardaron en ficheros de textos para luego graficarlos. Se le fue variando el estilo de construcción a los grafos programados, haciendo aristas de distintos colores y grosores, así como el tamaño, forma y color de los nodos. Este trabajo permitió desarrollar distintos tipos de grafos y familiarizarse con el lenguaje de programación **Pythonn** y la herramienta **Gnuplot**.

La tarea fue entregada en tiempo y calificada con 9, debido a errores de ortografía y redacción. Seguidamente se muestra el informe de la **Tarea 1** revisada por la profesora.

Tarea 1 de Optimización de Flujo de Redes

Evelly Gutiérrez Noda, ~~Matrícula 1005050~~

February 15, 2018

1 Introducción

En el siguiente reporte se abordará el tema estudiado en la clase de Optimización de Flujo de Redes impartida por la Dra. Elisa. En dicha clase estudiamos sobre la construcción de grafos utilizando lenguaje Python para la programación cuando queremos crear un grafo, además utilizamos la herramienta Gnuplot, la cual vinculamos con Python para representar el grafo antes programado.

2 Teoría sobre los Grafos

Primeramente, conocimos un poco sobre los grafos, los cuales son un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas, que pueden ser orientados o no. También sabemos que un grafo se representa mediante una serie de puntos (nodos o vértices) conectados por líneas (aristas), ver figura.

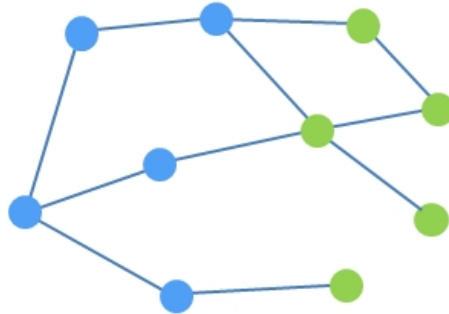


Figure 1: Ejemplo de Grafo.

Existen distintos tipos de grafos como por ejemplo:

- **Grafo simple:** Es el grafo donde una arista cualquiera es la única que une dos vértices específicos.
- **Multigrafo:** Es el que acepta más de una arista entre dos vértices. Los grafos simples son una subclase de esta categoría de grafos. También se les llama grafos en general.
- **Pseudografo:** Son aquellos que incluyen algún lazo.

- **Grafo orientado:** Es ~~un~~ grafo dirigido. Son grafos en los cuales se ha añadido una orientación a las aristas, representada gráficamente por una flecha.
- **Grafo etiquetado:** Grafos en los cuales se ha añadido un peso a las aristas (número entero generalmente) o un etiquetado a los vértices.
- **Grafo aleatorio:** Grafo cuyas aristas están asociadas a una probabilidad.
- **Hipergrafo:** Grafos en los cuales las aristas tienen más de dos extremos, es decir, las aristas son incidentes a ~~2~~ ^{3 o más} o más vértices.
- **Grafo infinito:** Son aquellos grafos con un conjunto de vértices y aristas de cardinal infinito.
- **Grafo plano:** Los grafos planos son aquellos cuyos vértices y aristas pueden ser representados sin ninguna intersección entre ellos.
- **Grafo regular:** Un grafo es regular cuando todos sus vértices tienen el mismo grado de valencia.

3 Descripción del trabajo en clase

En la clase aprendimos como programar en Python un grafo a partir de una cantidad de nodos **n**, tomando por ejemplo una cantidad de nodos **n = 10**, luego utilizando un ciclo **for** creamos un conjunto de nodos con valores aleatorios, los imprimimos en la consola y luego los almacenamos en un archivo de texto llamado **nodos.dat**. Para ello utilizamos las sentencias de código siguientes en la consola de Python:

```
from random import random
x = random()
y = random()
n = 10
for nodo in range(n):
    print(random(),random())
    with open("nodos.dat","w") as archivo:
```

\$n\$

~~~~~

▷ Variables para almacenar los nodos

▷ Cantidad de nodos

▷ Ciclo para imprimir los 10 nodos

▷ Trabajo con archivo nodos

Luego de tener los nodos creados, construimos las aristas correspondientes de nodo a nodo. Estas aristas las almacenamos en otro fichero llamado **aristas.dat**. Este trabajo se realizó en la consola de python, donde utilizamos dos ciclos **for**, uno para recorrer los nodos guardados anteriormente y el otro para agregar un nodo tras del otro siguiendo una condición especificada en un **if**. A continuación esta el código utilizado para este trabajo.

```
with open("aristas.dat","w") as aristas:
    for (x1,y1) in nodos:
    for (x2,y2) in nodos:
        if random() < 0.1:
            print(x1,y1,x2,y2,file= aristas)
```

for

for

▷ Trabajo con archivo aristas

▷ Iteración sobre los primeros pares de coordenadas

▷ Iteración sobre los segundos pares de coordenadas

▷ Condición para graficar

▷ Imprime en consola las coordenadas

Luego del trabajo en consola del Python, aprendimos a trabajar vinculando el Python con Gnuplot, por medio de un editor de Python, donde combinamos los códigos de cada programa con una secuencia lógica, para luego abrirlo desde el Gnuplot y de este modo quedaría representado el grafo antes programado (ver figura 2).

Esta representación del grafo en el editor de Python se hace utilizando los comandos de Gnuplot **"set arrow ..."** para representar las aristas como "flechas", en este comando también definimos cuales serían los nodos a unir y graficar y también se puede especificar el estilo con que se va a graficar, en la clase lo hicimos con el estilo que trae por defecto el comando.

Para representar el gráfico de puntos desde el archivo **nodos.dat**, por medio de este editor de Gnuplot con comandos de Python, usamos **"plot 'nodos.dat' using 1:2"**.

```

grafo2.py - C:\Users\SAMUNG\AppData\Local\Programs\Python\Python36-32\grafo2.py (3.6.4)
File Edit Format Run Options Window Help
print("hola")
from random import random
nodos = []
n = 10

with open("nodos.dat", "w") as nodo:
    for nod in range(n):
        x = random()
        y = random()
        nodos.append((x,y))
        print(x,y,file=nodo)

with open("grafo.plt", "w") as aristas:
    print("set xrange [-0.1:1.1]", file = aristas)
    print("set yrange [-0.1:1.1]", file = aristas)
    num = 1
    for (x1,y1) in nodos:
        for (x2,y2) in nodos:
            if random() < 0.1:
                print("set arrow {i:d} from {i:f}, {i:f} to {i:f}, {i:f} nohead".format(num,x1,y1,x2,y2),file= aristas)
                num += 1

    print('set size square', file = aristas)
    print('set key off', file = aristas)
    print('set xrange[0:1]', file = aristas)
    print('set yrange[0:1]', file = aristas)
    print("plot 'nodos.dat' using 1:2 with points pt 7", file = aristas)

print("termino")
Ln: 7 Col: 24

```

Handwritten blue annotations:

- A large blue arrow points from the text "set arrow ..." in the first paragraph to the corresponding line in the code.
- A blue circle highlights the word "nodos" in the text "desde el archivo nodos.dat".
- A blue circle highlights the text "plot 'nodos.dat' using 1:2" in the second paragraph.
- The word "Listings" is handwritten in blue ink.
- A blue arrow points from the word "Listings" to the code block.

Figure 2: Editor de Gnuplot para trabajar con Python

Como los valores de los nodos los creamos aleatorios cada vez que se corre el programa en el editor de Python, se crean nodos diferentes, que arrojaron la creación de diferentes grafos, a partir de varios valores de $n = 5, 6, 10, 100$ que definen la cantidad de nodos. (ver figura 3)

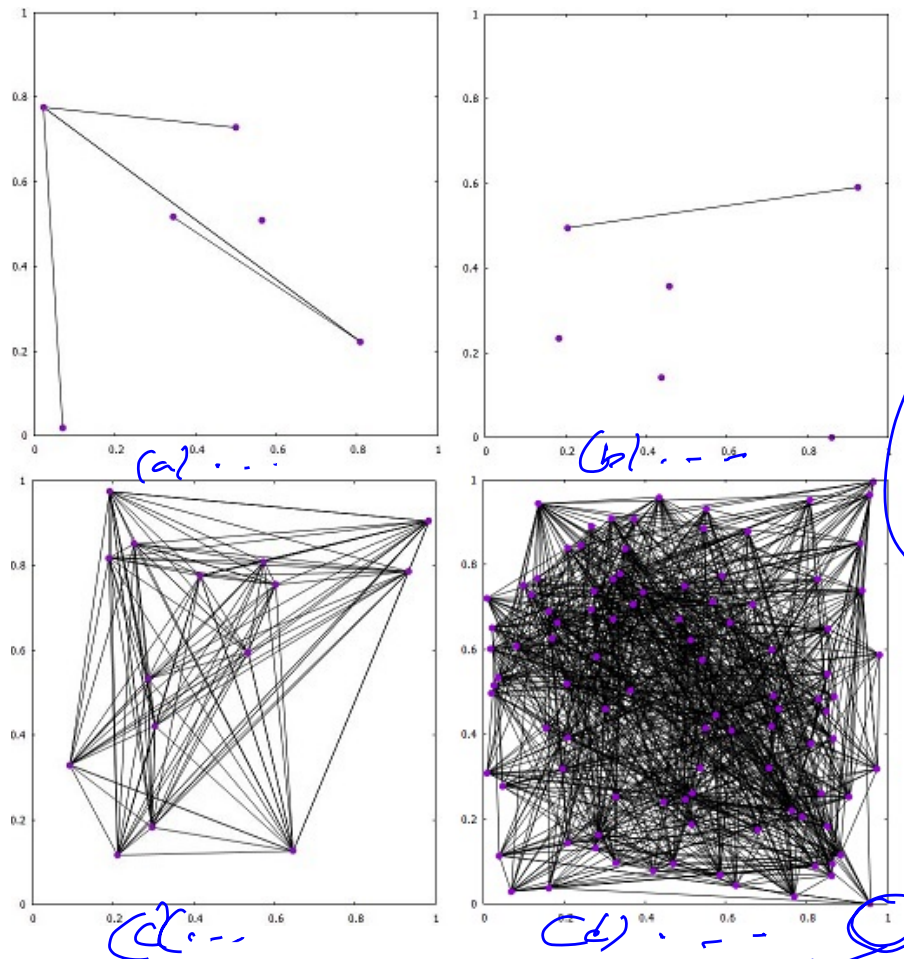


Figure 3: Grafos resultantes de lo aprendido en clase

4 Tarea Extraclase

Hasta aquí fue lo aprendido en clase, luego de forma investigativa individual, se hicieron varias modificaciones al grafo programado, cambiando la forma de los nodos, ya sean triángulos, cuadrados y círculos, y cambiando de color y grosor las aristas que unen los nodos. También se le puso nombres a los ejes del plano XY y nombre al grafo a representar. (ver figura 4)

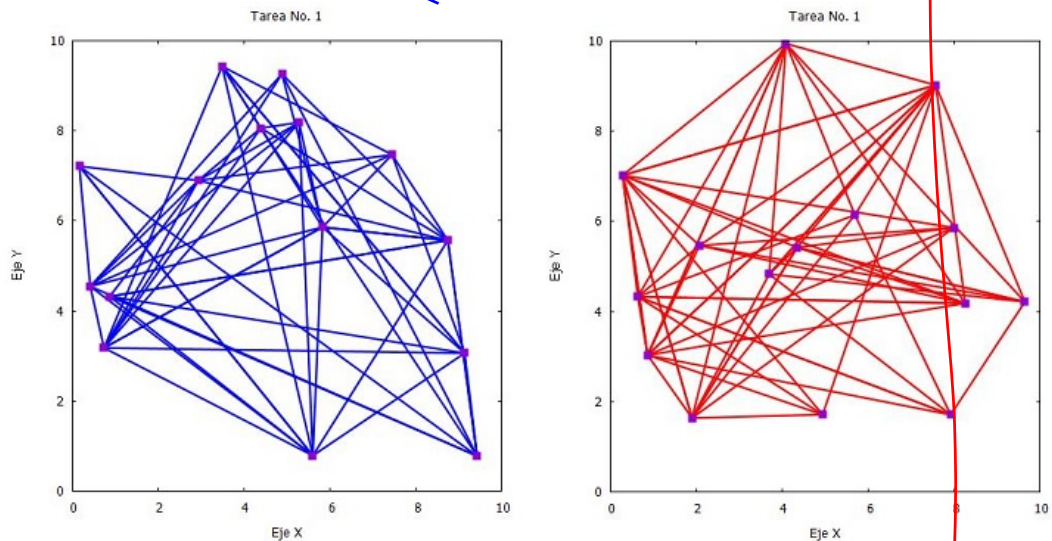


Figure 4: Cambios de grosor y color de aristas

Para colocar el nombre al eje X y Y, se utilizó el comando `set xlabel "eje x"` y `set ylabel "eje y"` respectivamente, para el título del grafo se utilizó el comando `set title (t) "Título"`.

El cambio de color y de grosor de las aristas fue utilizando los comandos `linetype n (lt n)` y `linewidth g (lw g)` respectivamente.

Con esto se puede ver la aplicación del uso de grafos para poder resolver o representar problemas de flujo en la vida real, ya sea de transporte, servicio de red o cualquier problema en el cual se desee conocer la vía más económica para resolverlo.

Un ejemplo hipotético podría ser para describir o representar las Líneas del metro que existen en Monterrey, utilizando lo aprendido, representar de diferentes colores el flujo de cada línea del metro, y los nodos serían las estaciones correspondientes. Quedarían las líneas rojas para representar la línea del metro 1, y la azul, para la línea del metro 2. (ver figura 5)

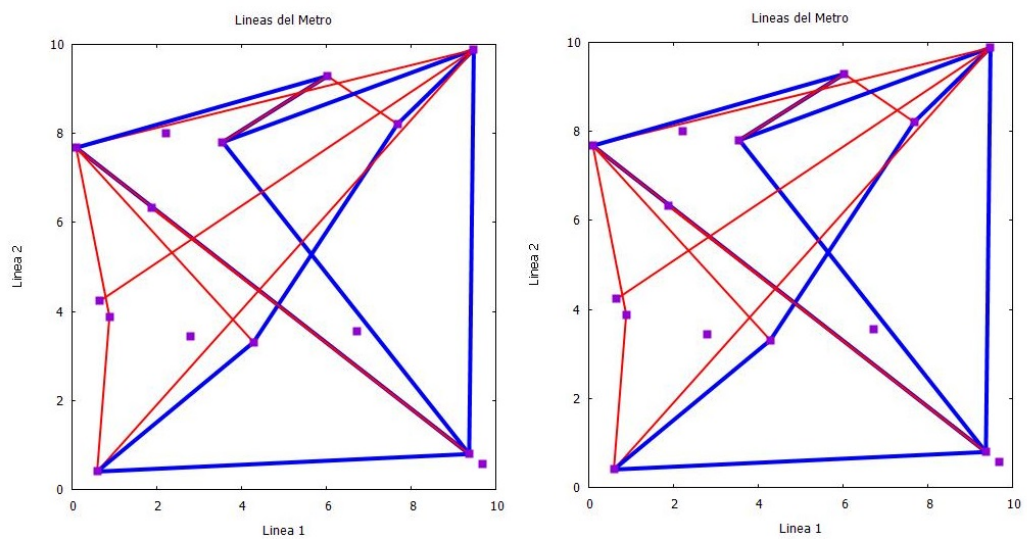


Figure 5: Líneas del Metro

1. Discusión de la Tarea 2

La **Tarea 2** se estudió el trabajo con grafos Simples, Dirigidos y Ponderados. Se utilizó el grafo programado en la tarea anterior para convertirlo en un grafo además de simple, dirigido y ponderado.

Se utilizó para este trabajo las estructuras de clases en **Python**, creando la clase Grafo para donde se definieron los parámetros correspondientes al grafo que se desee programar y las funciones para las operaciones que se le realizaron al grafo. Todo esto se realizó con el objetivo de organizar el código y de esta forma hacerlo general para la programación de cualquier grafo que se desee.

Se le agregó peso a las aristas que conectan los nodos del grafo, convirtiéndolo en un grafo ponderado, y luego se le agregó direcciones a las mismas, para convertirlo en dirigido, y de este modo trabajar con los tres tipos de grafos, simples, ponderados y dirigidos.

Se entregó en tiempo la tarea y recibió una calificación de 9 debido a algunos errores ortográficos y algunas imprecisiones de diseño y estilo del texto del informe.

A continuación se muestra el informe de correspondiente a la **Tarea 2** revisado por la profesora.

Tarea 2 Optimización de Flujo de Redes

Evely Gutiérrez Noda

4 de marzo de 2018

1. Introducción

En el siguiente reporte se abordará el tema estudiado en la clase de Optimización de Flujo de Redes. En dicha clase continuamos el estudio sobre la construcción de grafos, utilizando lenguaje Python [1] para la programación cuando queremos crear un grafo, además utilizamos la herramienta Gnuplot [2], la cual vinculamos con Python para representar el grafo antes programado.

Esta vez en la clase se estudió el trabajo con grafos Simples, Dirigidos y Ponderados, cuyas definiciones se dan a continuación.

Grafo Simple

Es un grafo donde una arista cualquiera es la única que une dos vértices específicos.

Grafo Dirigido

Son grafos en los cuales se ha añadido una orientación a las aristas, representada gráficamente por una flecha.

Grafo Ponderado

Grafos en los cuales se ha añadido un peso a las aristas (número entero generalmente) o un etiquetado a los vértices.

Un grafo simple dirigido, se puede decir que es ponderado al atribuirle peso a cada arista.

2. Descripción del trabajo en clase

En la tarea anterior ya quedo programado un grafo con ciertas carecterísticas, ahora se utilizará este grafo que se creó, para convertirlo en un grafo ademas de simple, dirigido y ponderado.

Para realizar esta actividad comenzamos el trabajo con la estructura de clases en Python, la cual permitira crear metodos generales donde solamente se le cambia el valor a la cantidad de nodos y la probabilidad con que se unirán. De este modo se podrá representar distintos tipos de grafos.

Se creó la **clase Grafo**, donde se definieron los parametros n (nodos), x, y (vertice x y vertice y respectivamente), un conjunto E , donde más tarde se almacenará cada par de vértices que serán unidos por una arista, el color de esta arista y el peso correspondiente a la misma. También se creó el parámetro *destino*, donde se almacenará el nombre del archivo en el cual se guardan los nodos y aristas creados, este fragmento de código se muestra a continuación.

```
class Grafo:
```

```
def __init__(self):
    self.n = None # se crean las variables pero aun no se inicializan
    self.x = dict()
    self.y = dict()
    self.E = [] # conjunto vacio
    self.destino = None
```

Luego dentro de la **clase Grafo**, se comenzó a definir funciones para la programación de un grafo, partiendo de las características del grafo programado en la Tarea 1, pero en este caso estará dividido el código por clases, y dentro de sus clases estarán las funciones. Todo esto con el objetivo de organizar el código y de esta forma hacerlo general para la programación de cualquier grafo que se desee. A continuación se expone un fragmento de alguna de las funciones creadas dentro de la **clase Grafo**, y una breve explicación de cada una de ellas.

```
def creaNodos(self, orden): # creando los nodos
    self.n = orden # se asigna a n, el valor pasado por parametros
    for nodo in range(self.n):
        self.x[nodo] = random()*10 # da valores aleatorios
        self.y[nodo] = random()*10
```

Función creaNodos: En esta función, como su nombre lo indica, se van a definir el tamaño de n(nodos), y luego se va a dar valores aleatorios a las x y y correspondientes. Para definir una función se utiliza la palabra reservada **def**, y para hacer referencia a una variable global creada dentro de la clase donde se está trabajando, se utiliza la palabra reservada **self**.

Luego se crea la **función Imprimir** para guardar los nodos en un archivo determinado utilizando una estructura parecida a la antes explicada, donde se pasa por parámetro a la función, el archivo de destino donde se guardará, que es una de las variables que se definieron en la estructura de la **clase Grafo**, la cual es **self.destino = None**. El valor **None**, se utiliza cuando la variable se quiere crear vacía.

También se crea la **función Conecta** para conectar dos nodos por una arista, donde esta arista ahora tendrá un peso y un color definidos anteriormente. Luego se almacenará esta unión de dos nodos con una arista con color y peso, en el conjunto *E*, definido en la estructura inicial de la **clase Grafo**.

Función Conecta

```
def conecta(self, prob):
    for nodo in range(self.n - 1):
        for otro in range(nodo + 1, self.n):
            if random() < prob: # valor de probabilidad para unir de
                                # nodos
                peso = choice(pesos) # se cambian los pesos definidos
                color = choice(colores) #se cambian los colores
                if peso > 0:
                    self.E.append((nodo, otro, peso, color)) # se
                                                                # guarda en el conjunto E el par de nodos que
                                                                # van a ser unidos por una arista con un color y
                                                                # peso
                print(peso)
```

Estas variables **color** y **pesos**, se definen antes de crear la **clase Grafo**, y se le asignan varios valores a cada una, para que a la hora de crear las aristas éstas tengan distintos colores y distintos pesos. Estas variables se crean por medio de las siguientes líneas de código justo antes de crear la **clase Grafo**.

```
colores = ["black", "blue", "pink", "orange", "red"]
pesos = [1,2,3,4, 5, 6, 7, 8]
```

Al ponerle un peso a cada arista esta operación convierte al grafo programado de grafo simple a grafo ponderado.

Luego se creó la **función Grafica**, donde se recorre el archivo con las aristas que ya se crearon anteriormente entre los nodos, y se plotean en el programa **Gnuplot**. Pero en este caso, en el **set arrow** (que se explicó en la tarea 1) donde se grafica la arista, esta vez se define que la arista tenga dirección, utilizando el parámetro **head**.

Al ponerle dirección a las aristas del grafo, se convierte de grafo simple a grafo dirigido, y como ya tiene peso en las aristas también es un grafo ponderado.

Luego de tener la **clase grafo** creada, con las funciones necesarias, entonces se crea otro fichero donde solamente se llamará a cada función, con los valores que se deseen, para construir el grafo (cantidad de nodos n , la probabilidad con que se conectarán los nodos $prob$ y los nombres de archivos donde se almacenarán los nodos y aristas), haciendo de este modo un programa general para crear grafos, partiendo de los parámetros que se le deseen pasar, en el ejemplo que viene a continuación, se crea un grafo con $n = 20$ y $prob = 0,4$.

```
from grafo2 import Grafo
prueba = Grafo() # creo una variable con la estructura de la clase Grafo
prueba.creaNodos(20)
prueba.imprimir("prueba.txt")
prueba.conecta(0.4)
prueba.grafica("prueba.plot")
```

En las siguientes imágenes se ve el resultado de lo antes estudiado. En la figura 1, se ve un grafo dirigido, y en la figura ?? se ve un grafo ponderado y dirigido, en este caso la ponderación se refleja en el grosor de las aristas.

Referencias

- [1] Python Software Foundation, www.python.org/
- [2] Gnuplot, www.gnuplot.info

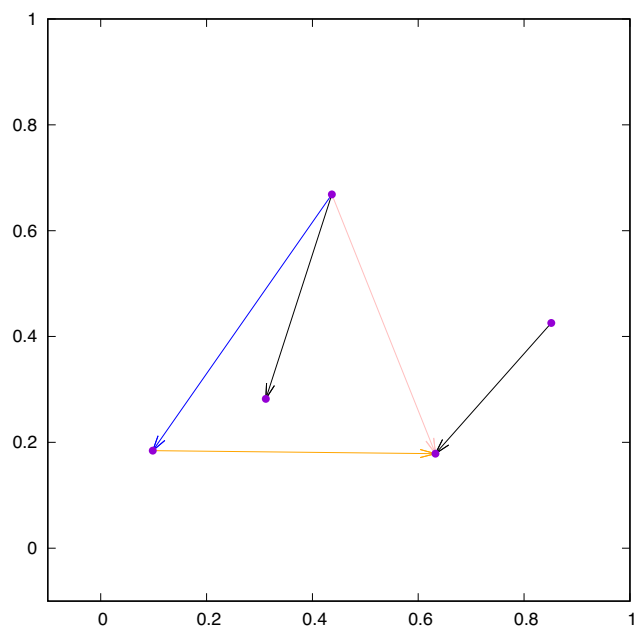


Figura 1: Grafo Dirigido.

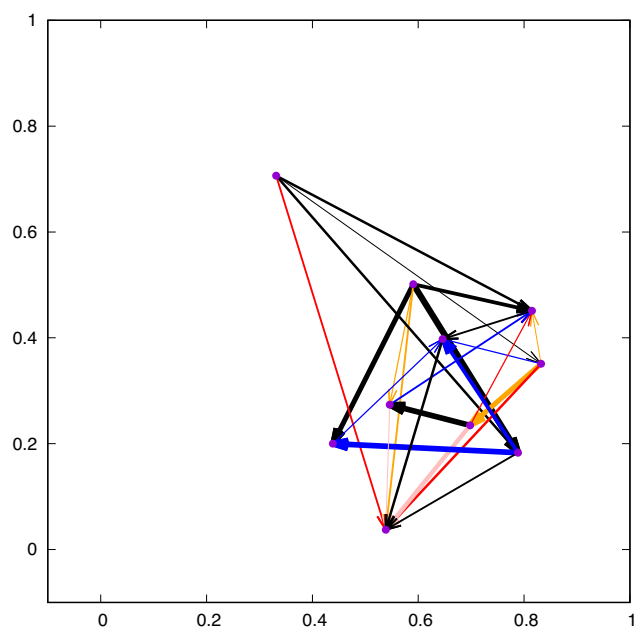


Figura 2: Grafo Ponderado y Dirigido.

1. Discusión de la Tarea 3

La **Tarea 3** se comenzó realizar experimentos con los algoritmos de **Floyd-Warshall** y **FordFulkerson**.

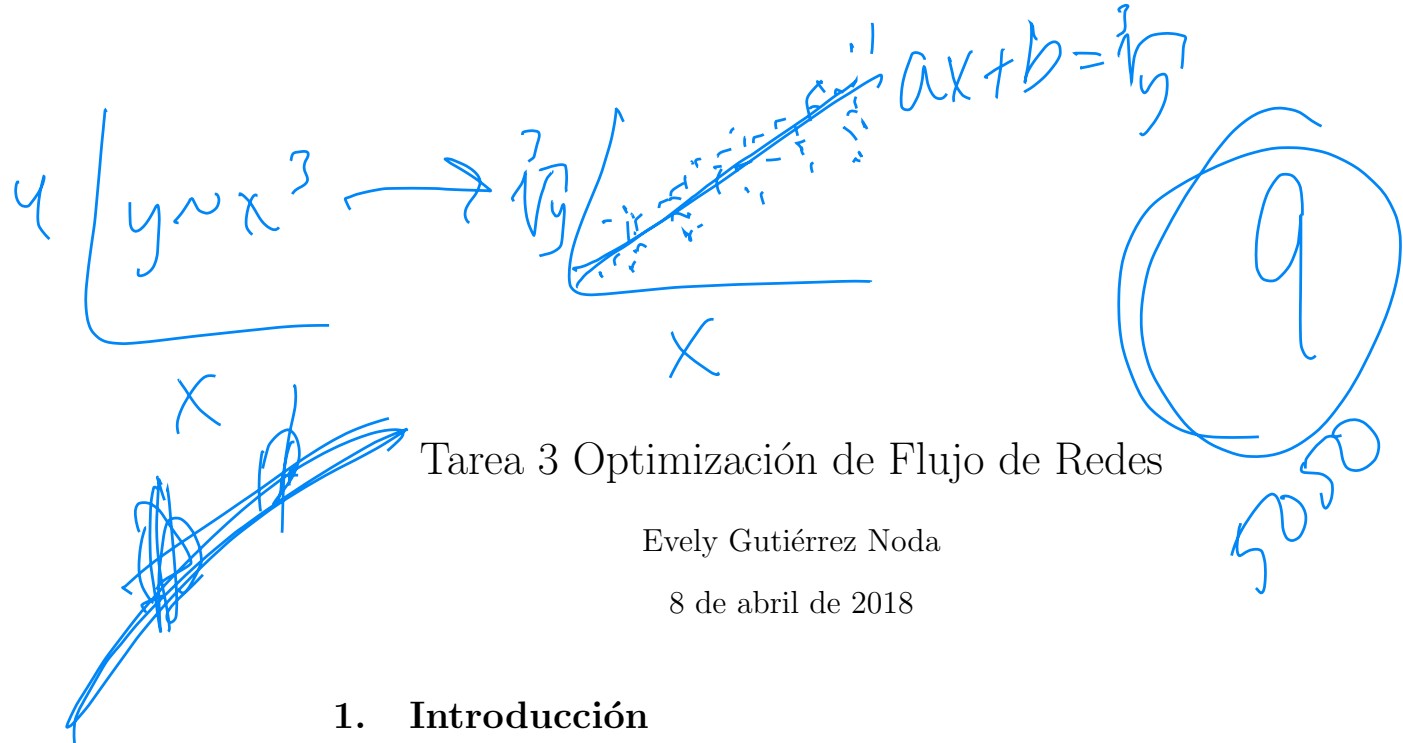
El algoritmo **FloydWarshall** es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. Su función es encontrar el camino entre todos los pares de vértices en una única ejecución. En cambio el algoritmo de **FordFulkerson** propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. La idea es encontrar una ruta con un flujo positivo neto que una los nodos origen y destino.

Se documentó el cambio que se realizó en el código para poder usar estos algoritmos. Además, se crearon ejemplos para verificar el funcionamiento correcto de estos algoritmos para grafos simples y ponderados, para el caso de los grafos dirigidos se verificó si funcionaban o no estos algoritmos.

Se crearon grafos de distintos tamaños para realizar estos experimentos de pruebas, basados en la estructura y definición para programar un grafo que se desarrolló en las tareas anteriores, utilizando el mismo lenguaje **Python** y la herramienta **Gnuplot** para graficar los resultados. Se probaron los algoritmos de **FloydWarshall** y **FordFulkerson** en estos grafos creados y se midió el tiempo de ejecución de cada uno y se representó este resultado en un diagrama de caja y bigote. Se pidió realizar el ajuste de curvas en el diagrama utilizado para representar los resultados de los tiempos.

Se entregó en tiempo la tarea y recibió una calificación de 9 debido a que no se logró en la tarea realizar el ajuste de curvas en el diagrama y a algunos errores en la gráfica donde se muestran los tiempos de ejecución de los algoritmos.

A continuación se muestra el informe de correspondiente a la **Tarea 3** revisado por la profesora.



Tarea 3 Optimización de Flujo de Redes

Evelly Gutiérrez Noda

8 de abril de 2018

1. Introducción

En el siguiente reporte se abordará el tema estudiado en la clase de Optimización de Flujo de Redes. En dicha clase continuamos el estudio sobre la construcción de grafos, utilizando lenguaje Python [?] para la programación cuando queremos crear un grafo, además utilizamos la herramienta Gnuplot [?], la cual vinculamos con Python para representar el grafo antes programado.

Esta vez en la clase se realizaron experimentos con el algoritmo de **Floyd-Warshall** y **FordFulkerson**.

2. Descripción del trabajo en clase

En la tarea anterior ya quedó programado un grafo con ciertas características, ahora se utilizará este grafo que se creó, para trabajar con el algoritmo **Floyd-Warshall** y **FordFulkerson**.

Partiendo de la **clase Grafo** que ya teníamos, donde se definieron los parámetros **n**(nodos), **x**, **y** (vértice **x** y vértice **y** respectivamente), un conjunto **E**, donde más tarde se almacenará cada par de vértices que serán unidos por una arista, el color de esta arista y el peso correspondiente a la misma. También se creó el parámetro **destino**, donde se almacenará el nombre del archivo en el cual se guardan los nodos y aristas creados.

Para el trabajo con estos algoritmos fue necesario crear en la estructura de la **clase Grafo**, un conjunto para almacenar los nodos vecinos a la hora de conectar cada uno de ellos con una arista, de modo que la **clase Grafo** quedó definida de la siguiente forma:

```
class Grafo:
```

```
    def __init__(self):
        self.n = None
        self.x = dict()
        self.y = dict()
        self.E = []
        self.destino = None
        self.vecinos = dict()#Nuevo conjunto para guardar los nodos
                               vecinos
        self.i = None
```

Luego dentro de la **clase Grafo**, que ya contenía funciones para la programación de un grafo, se agregaron los algoritmos de **FloydWarshall** para caminos cortos y **FordFulkerson** para flujo máximo. Se realizaron algunos cambios en el código utilizado en la Tarea 2 [[?]] para poder trabajar con estos algoritmos, ya que anteriormente no se trabajaba con un arreglo donde se guardarán los nodos vecinos, y este arreglo fue necesario crearlo para poder utilizar estos algoritmos como se mencionó antes.

El algoritmo **FloydWarshall** es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. Esto es semejante a construir una tabla con todas las distancias mínimas entre pares de ciudades de un mapa, indicando además la ruta a seguir para ir de la primera ciudad a la segunda. Este es uno de los problemas más interesantes que se pueden resolver con algoritmos de grafos.

Este algoritmo trabaja comparando todos los posibles caminos a través del grafo entre cada par de vértices, esto lo hace mejorando paulatinamente una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima. El algoritmo usa la metodología de Programación Dinámica para resolver el problema.

A continuación, se muestra como quedó modificado el algoritmo para poderlo utilizar en el grafo que se crea con el código de la Tarea 2. [[?]]

```
def FloydWarshall(self):
    d = {}
    for nodo in range(self.n - 1):
        d[(nodo, nodo)] = 0 # distancia reflexiva es cero
        for (vecino, peso) in self.vecinos[nodo]: # para vecinos, la
            distancia es el peso
            d[(nodo, vecino)] = peso
    for intermedio in self.vecinos:
        for desde in self.vecinos:
            for hasta in self.vecinos:
                di = None
                if (desde, intermedio) in d:
                    di = d[(desde, intermedio)]
                ih = None
                if (intermedio, hasta) in d:
                    ih = d[(intermedio, hasta)]
                if di is not None and ih is not None:
                    c = di + ih # largo del camino via "i"
                    if (desde, hasta) not in d or c < d[(desde,
                        hasta)]:
                        d[(desde, hasta)] = c # mejora al camino actual
    return d
```

El algoritmo de **FordFulkerson** propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. La idea es encontrar una ruta con un flujo positivo neto que una los nodos origen y destino.

Existe un flujo máximo que viaja desde un único lugar de origen hacia un único lugar de destino a través de arcos que conectan nodos intermediarios. Los arcos tienen una capacidad máxima de flujo, y se trata de enviar desde la fuente al destino la mayor cantidad posible de flujo.

Este algoritmo se utiliza para reducir los embotellamientos entre ciertos puntos de partida y destino en una red. por ejemplo: en sistemas de vías públicas, transporte de petróleo desde la refinería hasta diversos centros de almacenamiento, distribución de energía eléctrica a través de una red de alumbrado público, etc.

A continuación se muestra como quedó el algoritmo **FordFulkerson** luego de los cambios realizados para poderlo utilizar con las características de los grafos creados en la clase **Grafo**.

```
def FordFulkerson(self, s, t): # algoritmo de Ford y Fulkerson
    if s == t:
        return 0
    maximo = 0
    f = dict()
    while True:
        aum = self.camino(s, t, f)
        if aum is None:
            break # ya no hay
        incr = min(aum.values(), key = (lambda k: k[1]))[1]
        u = t
        while u in aum:
            v = aum[u][0]
            actual = f.get((v, u), 0) # cero si no hay
            inverso = f.get((u, v), 0)
            f[(v, u)] = actual + incr
            f[(u, v)] = inverso - incr
            u = v
        maximo += incr

    return maximo
```

Se realizaron varias pruebas con varios grafos de distintas características para medir el tiempo de ejecución de estos algoritmos. Estos grafos tenían una ponderación aleatoria con valores entre uno y diez. Uno de los ejemplos se realizó con un grafo ponderado **G1** con tamaño inicial de nodos (**n**) igual a 90, y el otro grafo ponderado **G2**, con tamaño inicial de nodos igual a 100. Los valores que arrojaron los algoritmos **FloydWarshall** y **FordFulkerson** se muestran en el resumen siguiente, donde los flujos máximos fueron **G1 = 275** y **G2 = 368**. Los tiempos de ejecución de estos algoritmos se midió en segundos, desde el instante en que fueron llamadas las funciones que los definen hasta que arrojaron un resultado.

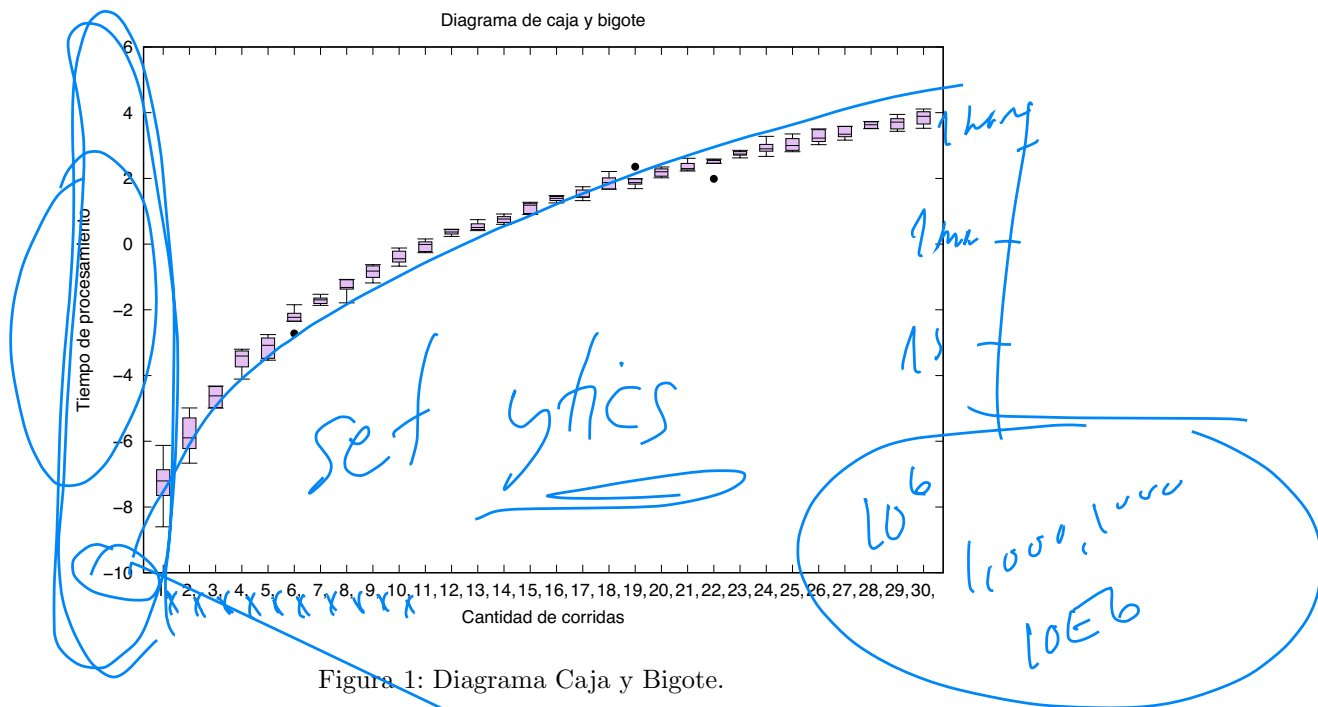
```
----- GRAFO 1 -----
Valores de N mas 5 :
90
prueba.txt
Resultado de FloydWarshall en cantidad de conjuntos :
8010
Tiempo de FloydWarshall:
0.7996770825935364
Tiempo de FordFulkerson:
4.868632634034739
FordFulkerson desde el inicio hasta n - 1
```

```

----- GRAFO 2 -----
Valores de N mas 5 :
120
prueba.txt
Resultado de FloydWarshall en cantidad de conjuntos :
14280
Tiempo de FloydWarshall:
1.8796005775237887
Tiempo de FordFulkerson:
15.671588563978176
FordFulkerson desde el inicio hasta n - 1
368

```

Este trabajo con los algoritmos se realizó con varios grafos de distintos tamaños y algunos del mismo tamaño, se fue aumentando en cinco en cada corrida el tamaño de los grafos, se midió el tiempo de ejecución para cada vez que se corrieron los algoritmos. Se realizaron pruebas de corridas de hasta treinta veces con tamaños de grafos que llegaron hasta ciento cincuenta, y en cada una se almacenaron los tiempos de ejecución para cada algoritmo en un fichero de texto que se utilizó para representar los resultados en un diagrama que se muestra en la figura ??.



Para grafos de igual tamaño, el experimento se realizó con grafos de sesenta nodos y se hicieron treinta corridas. Se guardaron los datos de los tiempos de ejecución de cada algoritmo en 5 veces que se realizó el experimento, y se graficaron los resultados de los tiempos en un diagrama que se muestra en la

figura ??.

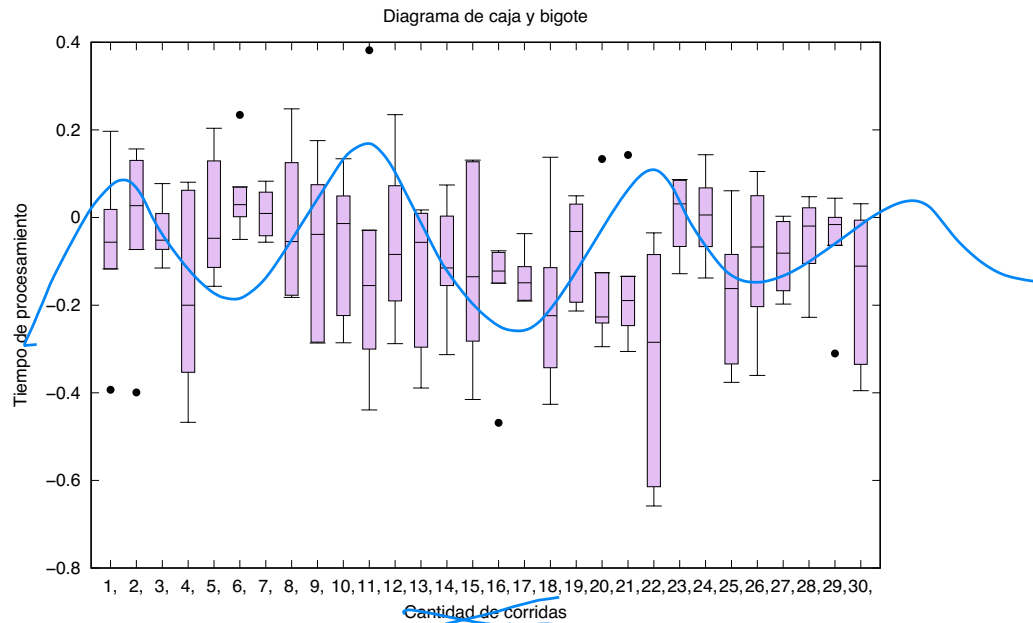


Figura 2: Diagrama Caja y Bigote.

Referencias

- [1] Python Software Foundation, www.python.org/
- [2] Gnuplot, www.gnuplot.info
- [3] Tarea2, <https://github.com/EvelyGutierrez/Optimizacion-de-flujo-de-redes/blob/master/Tarea2/VersionFinal/Tarea%202.pdf>

CLRS

1. Discusión de la Tarea 4

La **Tarea 4** se realizaron experimentos para crear grafos circulares conectando sus aristas con una probabilidad **p** y en dependencia de un valor **k** para la cantidad de conexiones que tendrá cada nodo, de este modo se conecta cada nodo con el que le sigue y así sucesivamente, quedando un grafo en forma de flor. Los nodos se crearon partiendo de un centro definido y se tuvo en cuenta la distancia de los nodos al centro por un radio especificado.

Se trabajó además las funciones **avgdist** y **clustcoef2** para calcular el promedio de la distancia, para la cual se trabajó con el algoritmo creado en tareas anteriores, **FloydWarshall**, y la otra función realiza el cálculo de la densidad del grafo, la cual se calculó tomando el número de aristas entre el número de aristas máximos que podría tener un nodo, de modo que todos los nodos queden conectados con todos.

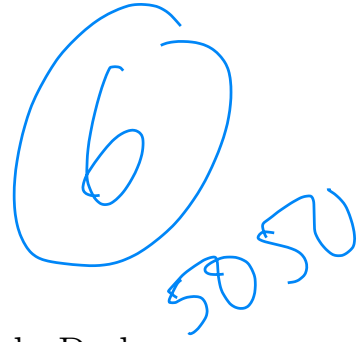
Se pidió establecer una cota superior respecto a la distancia máxima entre los nodos.

Se crearon varios grafos de distintos tamaños y algunos del mismo tamaño, se fue aumentando en cinco en cada corrida el tamaño del grafo y se midió el tiempo de ejecución para cada vez que se corrían las funciones creadas y los grafos construidos. Se realizaron pruebas de corridas de hasta diez veces con tamaños de grafos que llegaron hasta **n = 400**, y se representaron los tiempos de ejecución para cada función en ficheros de texto que se utilizaron para representar los resultados en un diagrama.

Se realizaron más experimentos trabajando con las mismas funciones y variando el parámetro **k** y la probabilidad **p**, los valores de **k** variaron desde uno hasta la cantidad de corridas, es decir, hasta diez en algunas ocasiones. Los resultados de este otro experimento se representaron en un diagrama de tres ejes que describe la variación de la distancia normalizada (calculada por la función **avgdist**), la variación de la densidad promedio (calculada por la función **clustcoef2**) y las probabilidades con que se trabajó.

Se entregó en tiempo la tarea y recibió una calificación de 6 debido a que no se logró una buena representación de los resultados de los experimentos descritos en los diagramas. Además, se rectificaron algunas precisiones en el código para mejorar la eficiencia.

A continuación se muestra el informe de correspondiente a la **Tarea 4** revisado por la profesora.



Tarea 4 Optimización de Flujo de Redes

Evely Gutiérrez Noda

22 de abril de 2018

1. Introducción

En el siguiente reporte se abordará el tema estudiado en la clase de Optimización de Flujo de Redes. En dicha clase continuamos el estudio sobre la construcción de grafos, utilizando lenguaje Python [1] para la programación cuando queremos crear un grafo, además utilizamos la herramienta Gnuplot [2], la cual vinculamos con Python para representar el grafo antes programado.

Esta vez en la clase se realizaron experimentos para crear grafos circulares conectando sus aristas con una probabilidad p y en dependencia de un valor k para la cantidad de conexiones. Se trabajó además con dos funciones para calcular el promedio de las distancias de las densidades de las vecindades de los nodos y para calcular la densidad del grafo.

2. Descripción del trabajo en clase

Primeramente se creó un grafo partiendo de una cantidad de nodos n , estos nodos se conectarán con los otros en dependencia de un valor k , el cual indicará la cantidad de conexiones que tendrá un nodo y lo conectará con el nodo que le sigue y así sucesivamente. Los nodos se crearán partiendo de un centro definido (se tomó como centro $(0.1 \text{ y } 0.5)$). Además se tuvo en cuenta la distancia de los nodos al centro por un radio (se tomó como radio 0.5). A continuación se muestra el código de la función programada para la creación del grafo antes descrito.

```
def GuardaCirculo(self, dest, k, N, prob, r, xo, yo): #Para crear
    vertices que formen un circulo y guardarlos en un .txt
    self.destino = dest
    self.i = 1
    size = random()
    with open(self.destino, "w") as circulo:
        for n in range(N):
            xn = xo + r * (math.cos(2*math.pi * (n/N)))
            yn = yo + r * (math.sin(2*math.pi * (n/N)))
            print(xn, yn, file = circulo)
            self.x[n] = xn
            self.y[n] = yn
            self.agrega(size, (xn, yn))

        for a in range(N):
```

```

for j in range(k):
    jj = (a+j) % n
    self.ConectaAristas(str(a), str(jj))
    self.i += 1

for i in range(n-1):
    for j in range(n-2*k-1):
        jj = (i+j+k+1) % n
        if random() < prob:
            self.ConectaAristas(str(a), str(jj))

return self.E

```

Handwritten notes: $\text{range}(1, k+1)$ (pointing to the first loop), (i, n) (pointing to the second loop), and a large scribble on the right.

El grafo que quedó como resultado se muestra en la figura 1.

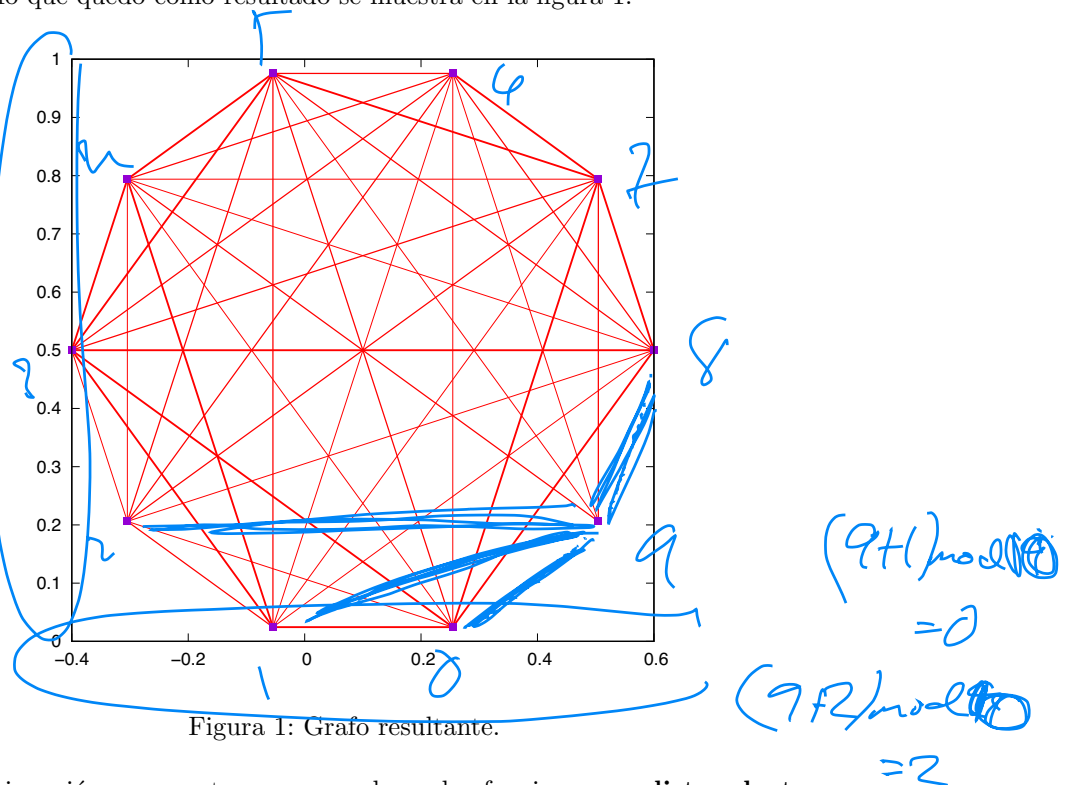


Figura 1: Grafo resultante.

A continuación, se muestra como quedaron las funciones **avgdist** y **clust-coef2** para calcular el promedio de la distancia, para la cual se trabajó con el algoritmo creado en la Tarea 2 [[3]], **FloydWarshall**, y la otra función es para el cálculo de la densidad del grafo, la cual se calculó tomando el número de aristas entre el número de aristas máximos que podría tener un nodo, de modo que todos los nodos queden conectados con todos.

```

def avgdist(self): #Promedio de las distancias
    self.d = self.FloydWarshall()
    self.sumatoria = sum(self.d.values()) / len(self.d)
    return self.sumatoria

```

```
def clustcoef2(self): # Densidad del grafo

    g = len(self.vecinos2) - 1
    valor = 0
    for v in range(1, g):
        m = 0
        for u in self.vecinos2[str(v)]:
            for w in self.vecinos2[str(v)]:
                if u in self.vecinos2[str(w)]:
                    m+= 1
        n = len(self.vecinos2[str(v)])
        if n > 1:
            valor += m/(n*(n-1))
    return(valor/g)

def ver(self):

    a = self.vecinos
    return a
```

Este trabajo se realizó con varios grafos de distintos tamaños y algunos del mismo tamaño, se fueron aumentando en cinco en cada corrida el tamaño de **n** en los grafos, se midió el tiempo de ejecución para cada vez que se corrían las funciones creadas y los grafos construidos. Se realizaron pruebas de corridas de hasta diez veces con tamaños de grafos que llegaron hasta **n = 400**, y en cada una se almacenaron los tiempos de ejecución para cada función en ficheros de texto que se utilizaron para representar los resultados en un diagrama que se muestra en la figura 2.

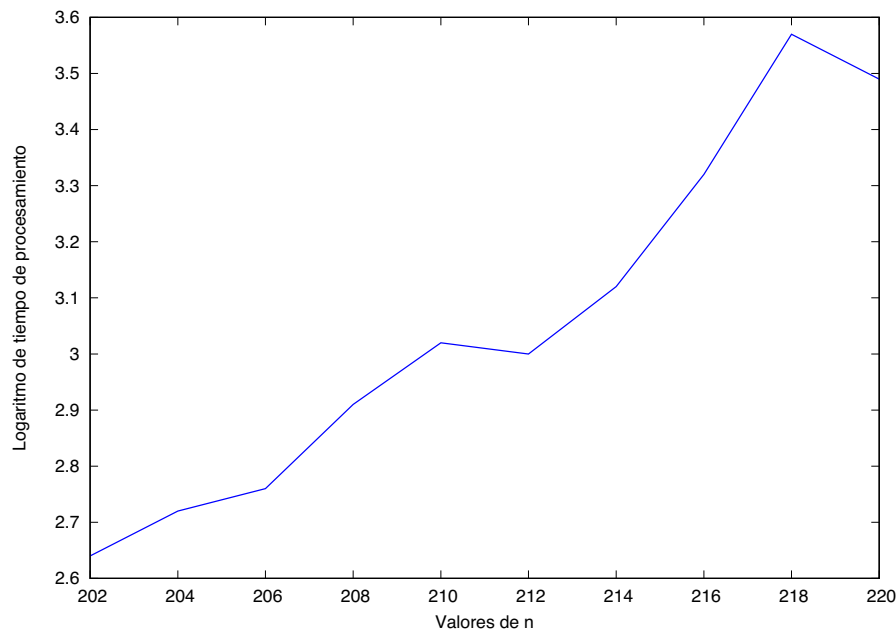


Figura 2: Diagrama de tiempos de ejecución.

Luego se realizaron más experimentos trabajando con estas funciones anteriormente descritas y variando el parámetro k y la probabilidad p , ambos para establecer las conexiones entre los nodos, los valores de k variaron desde 1 hasta la cantidad de corridas que se hicieron, es decir, hasta diez en algunas ocasiones.

El gráfico de la figura 4, es un diagrama de dos ejes Y (Izquierdo y Derecho) [4], el **eje y (Izquierdo)** describe la variación de la distancia normalizada que se calcula por la función `avgdist`. En el **eje y (Derecho)** se grafica la variación de la densidad promedio calculada por la función `clustcoef2` y en el **eje x** se encuentran las probabilidades con que se trabajó.

Los valores de p variaron desde 0.1 hasta 1, y los resultados de los valores que arrojaron las funciones para este experimento se muestra en los diagramas de las figuras 3 y 4.

Referencias

- [1] Python Software Foundation, www.python.org/
- [2] Gnuplot, www.gnuplot.info
- [3] Tarea2, <https://github.com/EvelyGutierrez/Optimizacion-de-flujo-de-redes/blob/master/Tarea2/VersionFinal/Tarea%202.pdf>
- [4] Gráficos de 3 ejes, http://gnuplot.sourceforge.net/docs_4.2/node292.html

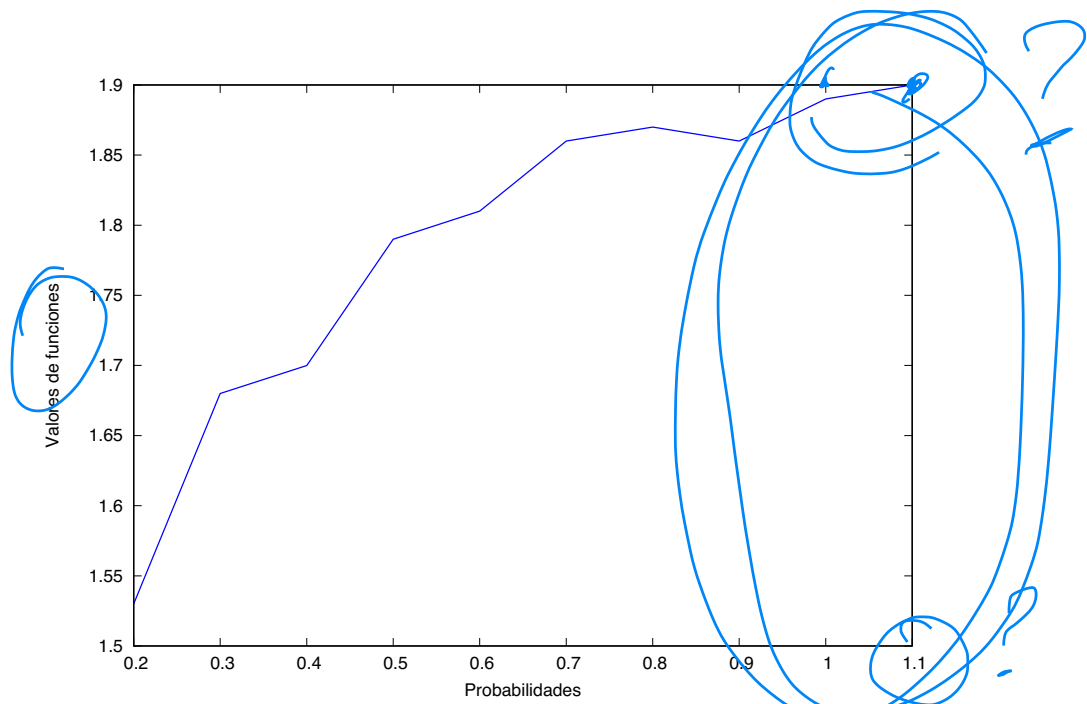


Figura 3: Diagrama de probabilidades contra funciones.

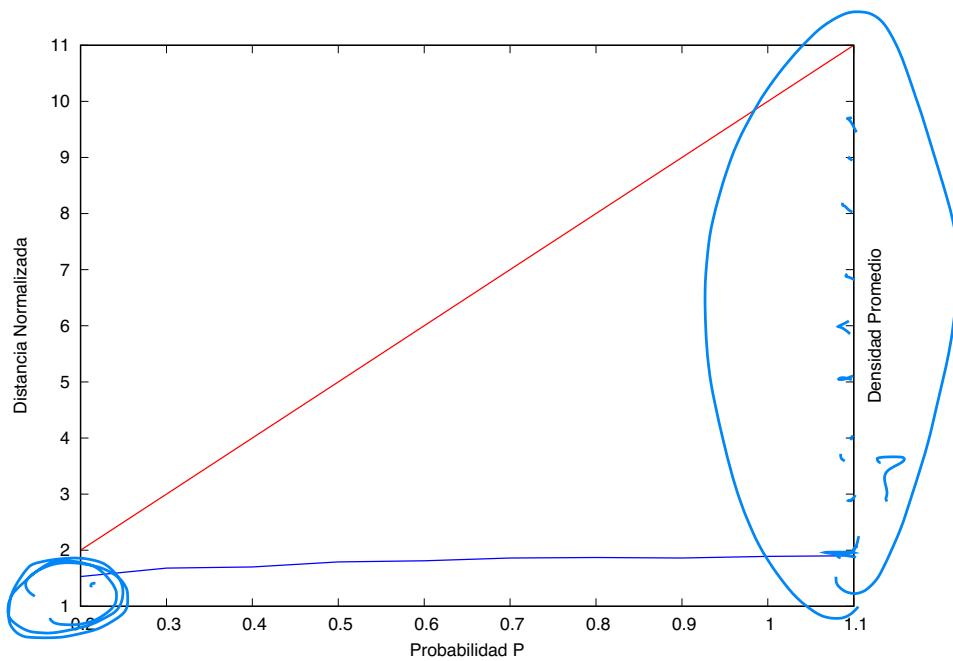
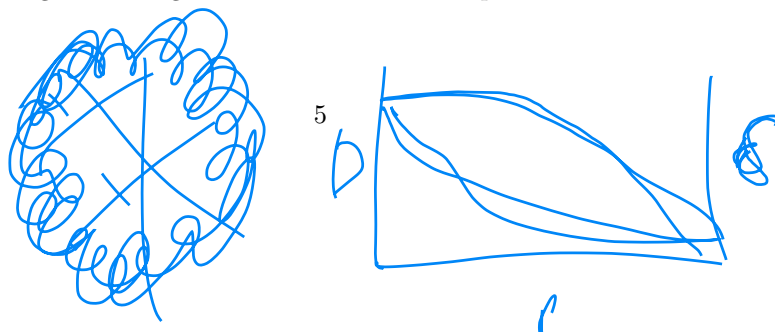


Figura 4: Diagrama de distancia contra probabilidad



1. Discusión de la Tarea 5

La **Tarea 5** se continuó el trabajo con el lenguaje Python y la herramienta para graficar Gnuplot. Se realizaron experimentos con un tipo de grafo en particular, estos grafos fueron contruidos basados en la **distancia de Manhattan** o la **geometría del taxista** como también suele llamarse.

Para el trabajo con esta distancia se crearon grafos a partir de varios parámetros que definen sus nodos, sus aristas y en la manera que se conectan. Uno de estos parámetros es el de distancia **L**, el cual especifica el número de conexiones que va a tener un nodo con otro, teniendo en cuenta que si **L = 0**, no existen conexiones en el grafo. Otro parámetro considerado fue **K**, el cual define la cantidad de vértices resultantes en el grafo, de modo que la cantidad de nodos sería igual a **K** al cuadrado.

Otro parámetro fue el de probabilidad **P**, con valores muy bajos no mayores que uno, para conectar aleatoriamente algunos nodos que no estén bajo la distancia de Manhattan (**L**), es decir que estarán conectados bajo la distancia euclidiana.

Otro propósito de esta tarea fue la programación de una función para eliminar nodos al azar del grafo, calcular el flujo máximo que corre por el grafo, y ver qué sucede con el mismo. Teniendo en cuenta que, al eliminar un nodo, también se eliminarán todas las aristas que estén conectadas a este nodo. Con estos cambios en el grafo se realizaron varias corridas para ver qué pasaba con el flujo máximo, si aumentaba, disminuía o se mantenía relativamente igual. Se realizaron gráficas para representar estos experimentos y el tiempo de ejecución de estas funciones.

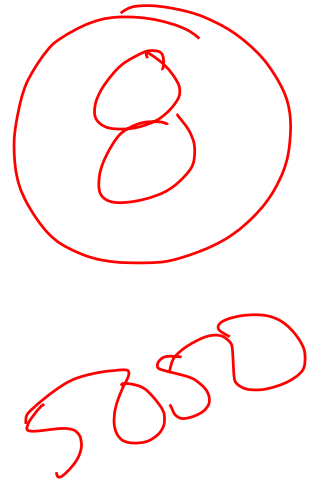
Se entregó en tiempo la tarea y recibió una calificación de 8 debido a que no se logró una buena representación de los resultados de los experimentos descritos en los diagramas, si se obtuvo resultados pero no quedaron representados en las gráficas.

A continuación se muestra el informe de correspondiente a la **Tarea 5** revisado por la profesora.

Tarea 5 Optimización de Flujo de Redes

Evely Gutiérrez Noda

6 de mayo de 2018



1. Introducción

En el reporte se abordará el tema estudiado en la clase de Optimización de Flujo de Redes. En dicha clase se continuó el estudio sobre la construcción de grafos, utilizando lenguaje Python [1] para la programación cuando queremos crear un grafo, además utilizamos la herramienta Gnuplot [2], la cual vinculamos con Python para representar el grafo antes programado.

En la clase se realizaron experimentos con un tipo de grafo en particular, estos grafos fueron contruidos basados en la **distancia de Manhattan** o la **geometría del taxista** como también suele llamarse.

2. Descripción del trabajo en clase

El trabajo realizado fue partiendo de un grafo cuyas características son similares a la geometría que describe la distancia de Manhattan. Esta es una forma de geometría en la que la métrica usual de la geometría euclidiana es reemplazada por una nueva métrica en la que la distancia entre dos puntos es la suma de las diferencias (absolutas) de sus coordenadas.

La **métrica del Taxista** también se conoce como **distancia rectilínea**, **distancia L**, **distancia de ciudad**, **distancia o longitud Manhattan**, con las correspondientes variaciones en el nombre de la geometría. El último nombre hace referencia al diseño en cuadrícula de la mayoría de las calles de la isla de Manhattan, lo que causa que el camino más corto que un auto puede tomar entre dos puntos de la ciudad tengan la misma distancia que dos puntos en la geometría del taxista de modo que el resultado de un camino desde un lugar a otro en esta ciudad quedaría como se muestra en la figura 1, donde las líneas azul y verde representan la distancia de Manhattan y la línea roja es para representar, en esa misma situación, la distancia euclidiana.



Figura 1: Distancia Manhattan y Distancia Euclidiana.

Para el trabajo con esta distancia se crearon grafos a partir de varios parámetros que definen sus nodos, sus aristas y en la manera que se conectan. Uno de los parámetros a considerar en este experimento es el parámetro de distancia \mathbf{L} , el cual especifica el número de conexiones que va a tener un nodo con otro, teniendo en cuenta que si $\mathbf{L} = 0$, no existen conexiones en el grafo.

Otro parámetro considerado fue \mathbf{K} , el cual define la cantidad de vértices resultantes en el grafo, de modo que si $\mathbf{K} = 5$, entonces la cantidad de nodos sería $\mathbf{N} = 25$, es decir \mathbf{K} al cuadrado.

Se trabajó con una valor de $\mathbf{K} = 5$ en principio, para poder visualizar correctamente las conexiones establecidas, luego este parámetro varió hasta 10. Por tanto la cantidad de nodos \mathbf{N} varió entre **25 y 100**. Además se utilizó un parámetro \mathbf{P} con valores muy bajos entre **0.0001 y 0.0010** para conectar aleatoriamente algunos nodos que no estén bajo la distancia de Manhattan (\mathbf{L}), es decir que estarán conectados bajo la distancia euclidiana. Como resultado de lo antes descrito se crearon los grafos que se muestran en las figuras 2, 3 y 4, donde las aristas de color negro indican las conexiones dependiendo del valor de \mathbf{L} , y las aristas de color azul, son las conexiones bajo la probabilidad de conexión \mathbf{P} .

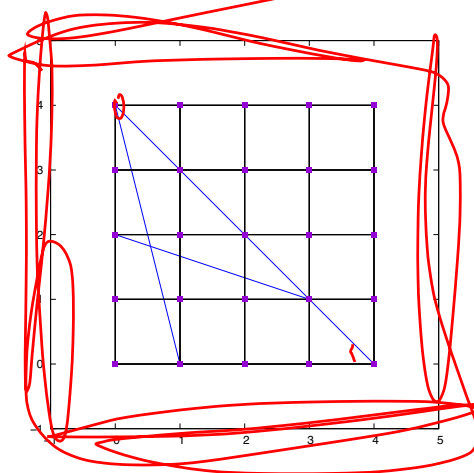


Figura 2: Grafo con $\mathbf{L} = 1$ y $\mathbf{P} = 0.008$

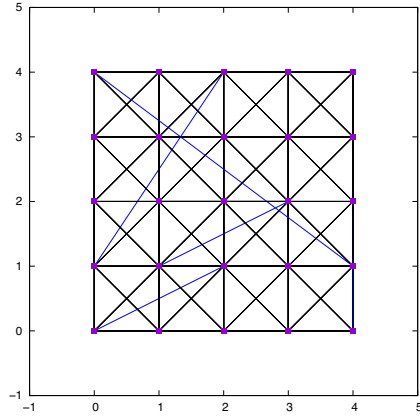


Figura 3: Grafo con $L = 2$ y $P = 0.008$

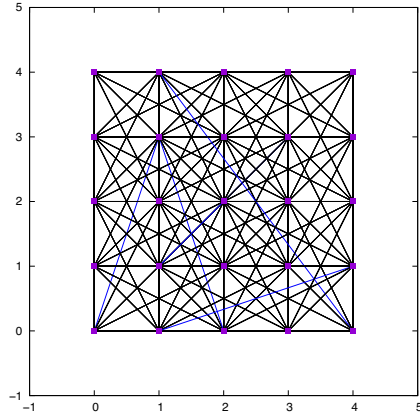


Figura 4: Grafo con $L = 3$ y $P = 0.008$

Para cada uno de estos grafos se calculó el flujo máximo, a medida que el valor de \mathbf{L} iba aumentando desde 1 hasta 10, de este modo se iban aumentando las conexiones entre los nodos, por lo cual iba aumentando a su vez el flujo máximo en el grafo creado. Este flujo máximo se calculó utilizando el algoritmo **FordFulkerson** descrito en la Tarea 3 [3]. Este proceso se describe en el diagrama que se muestra en la figura 5.

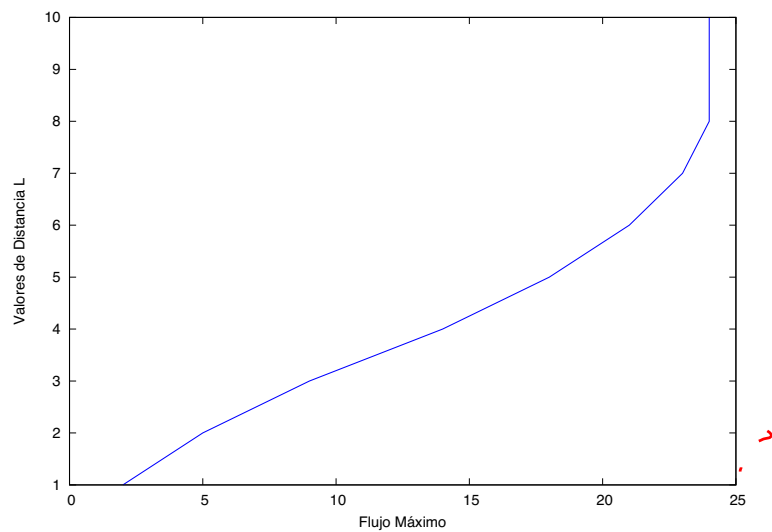


Figura 5: Distancia Manhattan L VS Flujo Máximo

Luego se realizó otro experimento, esta vez se programó una función para eliminar aristas al azar y de este modo, calcular el flujo máximo al quitar estas aristas, lo cual demostró que a pesar de quitar aristas el flujo máximo no vario notablemente, es decir, solo varió en pocas unidades.

Por último, se programó una función para eliminar nodos al azar del grafo, y como en el experimento anterior, calcular el flujo máximo que corre por el grafo, y ver qué sucede con el mismo. Al eliminar un nodo, también se eliminarán todas las aristas que estén conectadas a este nodo, de modo que el cambio en el flujo máximo puede ser más drástico, pero en este experimento el flujo se comportó sin muchas variaciones, indicando que el echo de que disminuyan las aristas o los nodos no influye directamente en la disminución del flujo máximo. También se midió el tiempo de ejecución del algoritmo utilizado para calcular el flujo máximo, el cual fue aumentando muy ligeramente. Algunos de los valores de como van disminuyendo los nodos y las aristas conectadas a estos nodos, se muestran en la salida de Python de la figura 6, así como también se ven los valores del flujo máximo cada vez que se elimina un nodo y sus aristas, y el tiempo de ejecución de este proceso completo.

Las funciones programadas para eliminar aristas y nodos al azar se muestran a continuación.

```
def EliminaArista(self, u,v):

    del self.aristas[u]
    del self.aristas[v]

    self.vecinos[u].remove(v)
    if not f:
        self.vecinos[v].remove(u)
```

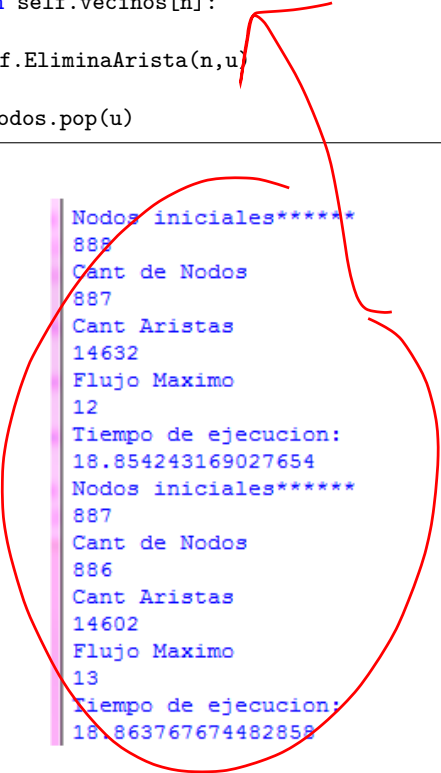
```
def EliminaNodo(self, u):

    vecino = self.vecinos[u].copy()
    for i in vecino:

        self.EliminaArista(u,i)
    for n in self.nodos:
        if u in self.vecinos[n]:

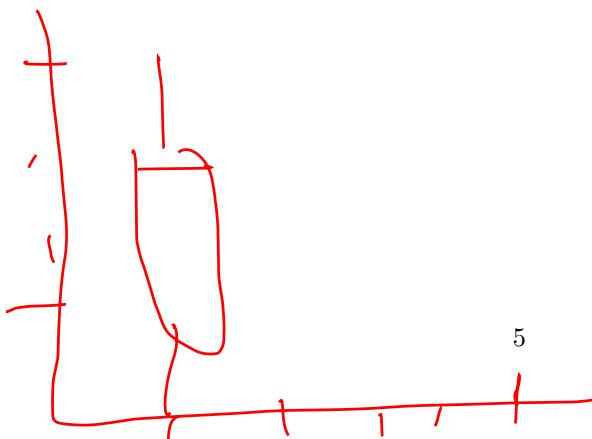
            self.EliminaArista(n,u)

    h = self.nodos.pop(u)
```



```
Nodos iniciales*****
888
Cant de Nodos
887
Cant Aristas
14632
Flujo Maximo
12
Tiempo de ejecucion:
18.854243169027654
Nodos iniciales*****
887
Cant de Nodos
886
Cant Aristas
14602
Flujo Maximo
13
Tiempo de ejecucion:
18.863767674482858
```

Figura 6: Salida de Python.



Referencias

- [1] Python Software Foundation, www.python.org/
- [2] Gnuplot, www.gnuplot.info
- [3] Tarea3, <https://github.com/EvelyGutierrez/Optimizacion-de-flujo-de-redes/blob/master/Tarea2/VersionFinal/Tarea%202.pdf>

1. Discusión de la Tarea 6

La **Tarea 6** se continuó el trabajo con el lenguaje Python y la herramienta para graficar Gnuplot.

Tuvo como objetivos principales programar un algoritmo de aproximación al corte mínimo, y por medio de este tener aproximaciones de cuál podría ser el flujo máximo en el grafo. Esto va a resultados diferentes, tomando el menor valor de los resultados, ya se tendría el valor del flujo máximo.

Otro experimento consistió en elegir aristas y contraerlas (unirlas), hasta llegar a tener dos únicos nodos conectados por todas las aristas. Con esta operación ya se tiene el corte mínimo.

Medir el tiempo de ejecución del algoritmo de **FordFulkerson** que es el utilizado para medir el flujo máximo en el grafo es otro de los objetivos de esta tarea. Además, graficar el tamaño que va teniendo el grafo mientras se unen los nodos contra los tiempos de ejecución del algoritmo para el flujo máximo y el algoritmo nuevo implementado para el corte mínimo programado que también arrojaría valores de flujos máximos.

Los experimentos de esta tarea se realizan sobre cualquier tipo de grafo, y se explica cuáles son las características del tipo de grafo seleccionado para trabajar.

Esta tarea no fue entregada debido a varios problemas que no hicieron posible dedicarle tiempo para realizarla. Fueron problemas del tipo personales y del tipo académicos, en lo personal fue una semana en la que mi hijo pequeño de 3 años estuvo enfermo y requería toda la atención posible, impidiendo una concentración correcta para realizar la tarea. En lo académico coincidió que además tenía otro proyecto a entregar en las mismas fechas y con prácticamente el mismo tiempo para entregar que la Tarea 6. Como buena estudiante y profesional que deseo ser pienso que, aunque los problemas que presenté en esta tarea no son justificantes para no realizarla, valoré la opción de que con las tareas entregadas ya tendría una calificación satisfactoria en la materia y decidí dedicarme a realizar el otro proyecto en los ratos que mi hijo descansaba y se sentía más aliviado.