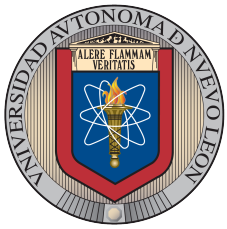


UANL

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

POSGRADO EN CIENCIAS EN INGENIERÍA DE SISTEMAS

OPTIMIZACIÓN DE FLUJO EN REDES

RETROALIMENTACIÓN Y AUTOEVALUACIÓN DE LAS TAREAS REALIZADAS DURANTE EL PERIODO FEBRERO-MAYO DE 2018

Beatriz Alejandra García Ramos

Matrícula: 1550385

Mayo de 2018

Tarea 1

Al realizar la actividad y tener una retroalimentación sobre el reporte se realizaron correcciones sobre los signos de puntuación en los títulos de la tarea y de cada sección que se escribió. Además de los signos que se encuentran en las imágenes realizadas para tener una visualización del grafo que se realizó en el código en `python`.

Además de tener correcciones en signos de puntuación se tienen también correcciones en la redacción y una corrección sobre escribir párrafos que no sean demasiado extensos ya que es algo pesado leer tanta información junta.

También se tienen sugerencias sobre el estilo de letra que se debe utilizar para ciertos conceptos y sobre la utilización de referencias dentro del texto.

Para que se puedan disminuir las palabras en los textos y no suceda que sean extensos se pueden incluir partes del código hecho en donde sea más clara la explicación de lo que sucedió en la práctica.

Tarea 1
Optimización de flujo en redes.

Beatriz Alejandra García Ramos

19 de Febrero de 2018

1. Introducción.

En esta práctica se desea enriquecer el conocimiento del programa `python` con un enfoque hacia la realización de grafos. Lo que se requiere es que mediante un código realizado en este programa se puedan generar archivos que después se procesen en el programa `gnuplot` que es un apoyo visual de nuestros resultados.

Un grafo puede ser o no dirigido, pero en esta práctica se hará un enfoque a grafos sin dirección, lo que se requiere es un código en el que se incluyan nodos que se conectarán mediante cierto criterio con algunos otros nodos, con uno, o con ninguno, dependiendo del criterio que se tome.

Lo que se desea realizar son dos tipos de grafos, el primero es un grafo cíclico y el segundo un grafo bosque, el cual es un conjunto de árboles y tiene como propiedad que los nodos busquen distancias cercanas a ellos.

2. Creación de un grafo en `python`.

Para la realización de los grafos principalmente se requieren nodos y aristas, los nodos son aquellos puntos en nuestro espacio que serán conectados o no dependiendo del criterio que se les indique. Las conexiones que se tienen entre ellos son las aristas, pueden ser de manera aleatoria o con alguna decisión que se indique en el código.

Para la práctica primero se realizaron pruebas en donde se tenía que los nodos eran conectados de manera aleatoria unos con otros, se indicaba una probabilidad p de existencia de una arista y era comparada con cifras aleatorias, si se cumplía el criterio de que la cifra aleatoria era menor a la probabilidad dada entonces los nodos comparados eran conectados por una arista.

Una vez que se realizó esa prueba se introdujeron y modificaron ciertas funciones. El proceso para realizar un grafo es primeramente crear los nodos en nuestro espacio, el cual es de dimensión 1×1, los nodos son creados con un ciclo `for` que nos ayuda a hacer tantos nodos como queramos, cada vez que se crea un nodo se va guardando en un archivo y el nodo se construye con dos variables, las cuales se toman de manera aleatoria un valor entre cero y uno y se hacen pares ordenados con estas variables. Una vez que tenemos un vector en donde se guardan estos nodos además de que se guarden en el archivo se hace el emparejamiento de los nodos, para los grafos que se tomarán en esta práctica se implementaron criterios distintos.

Para el primer tipo de grafo que es el cíclico se realizaron dos criterios, el primero fue que el último nodo creado debía ser conectado con el primer nodo para que cuando se hiciera el ciclo `for` que une al resto de los nodos fuera más sencillo realizar el trabajo. Una vez que se conectó el último nodo con el primero se llevó a cabo el ciclo `for` que determina que el primer nodo se conecta con el segundo nodo, el segundo con el tercero y así sucesivamente hasta llegar al nodo $n-1$, el cual se conecta con el nodo n. Éste es un tipo de grafo particular de los grafos cíclicos ya que se tiene que las conexiones son de manera secuencial.

Para el segundo tipo de grafo lo que se requiere hacer es encontrar para el último nodo el nodo más cercano a él y emparejarlo con éste. Después creando un ciclo `for` para cada uno de los nodos se buscan todas las distancias de los demás nodos a él, se localiza el mínimo de todas estas distancias

y se verifica que la arista creada entre estos dos nodos no se encuentre ya en la lista de aristas, si ya se encuentra se elimina la mínima distancia del nodo en proceso y teniendo en cuenta el resto de las distancias vuelve a buscar el mínimo y se crea la arista correspondiente, si no se encuentra la arista dentro de la lista entonces se tendrá solamente la instrucción de incluirla en ella. Sin embargo se tiene el problema de que pueden existir aristas repetidas, ya que en la decisión del código ir, por ejemplo, del nodo uno al nodo dos no es lo mismo que ir del nodos dos al uno, lo que ocasiona que esa arista se incluya dos veces ya que este grafo no es dirigido, por lo que se modificó el código de manera que no existieran repeticiones de aristas, para lo cual se realiza una manera distinta de abordar el problema: primero se calculan las distancias de todos los nodos hacia todos los nodos, es decir, se tendrán n^2 distancias y se colocan en una lista llamada `dis`, después nodo por nodo se calculan de nuevo las distancias hacia los otros nodos con un ciclo `for` y se tiene que en cada etapa si el mínimo de las distancias calculadas para el nodo en esa etapa se encuentra en la lista `dis` este nodo se conectará con el nodo correspondiente a la distancia calculada y ésta se eliminará de la lista `dis` para que cuando se realicen las siguientes etapas del ciclo `for` no se tome en cuenta la misma arista más de una vez, para el caso en que se calculen las distancias de de un nodo a sí mismo se tendría que la distancia es cero, entonces con el criterio que se está realizando las conexiones serían de los nodos consigo mismos, para evitar ésto se coloca un valor muy grande de la distancia en la lista `dis`.

3. Resultados.

Una vez que se realizó el código para el grafo cíclico se tenía que cuando la cantidad de nodos aumentaba no se veía con claridad cuáles nodos estaban conectados, por lo que se tomaron en cuenta solamente diez nodos para observar cómo se maneja el trabajo, como se muestra en la figura 1.

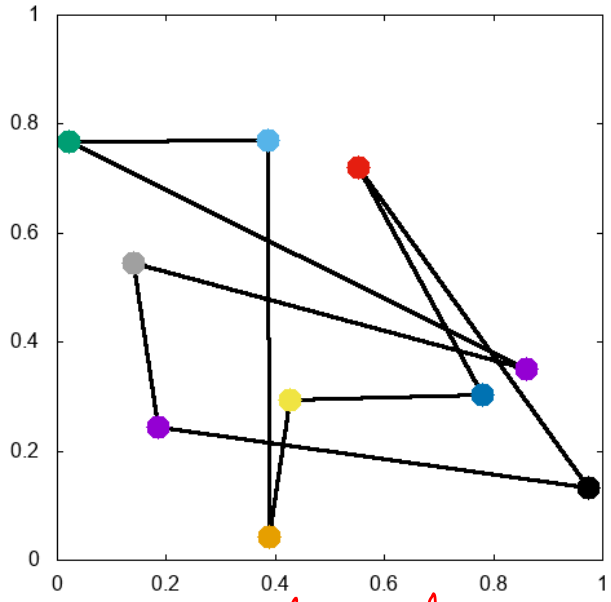


Figura 1: Grafo cíclico con 10 nodos.

Dado que el grafo cíclico está definido en el código con orden creciente se tienen los distintos tonos que `gnuplot` utiliza para el coloreo de los nodos los cuales se encuentran definidos en el cuadro 1. El color gris no se encuentra definido en el coloreo pero se localiza como el último nodo en el grafo.

Dado que no se tiene una buena visualización del grafo cíclico con los criterios que se tomaron en cuenta se decidió realizar los cambios del código para que se cumplieran otras características como el hecho de que se busquen los nodos cercanos y se conecten con ellos.

Por tanto, cuando se realizaron los cambios y se hizo la elección de los nodos cercanos se tomó en

Número	Color
1	morado
2	verde
3	azul celeste
4	naranja
5	amarillo
6	azul rey
7	rojo
8	negro

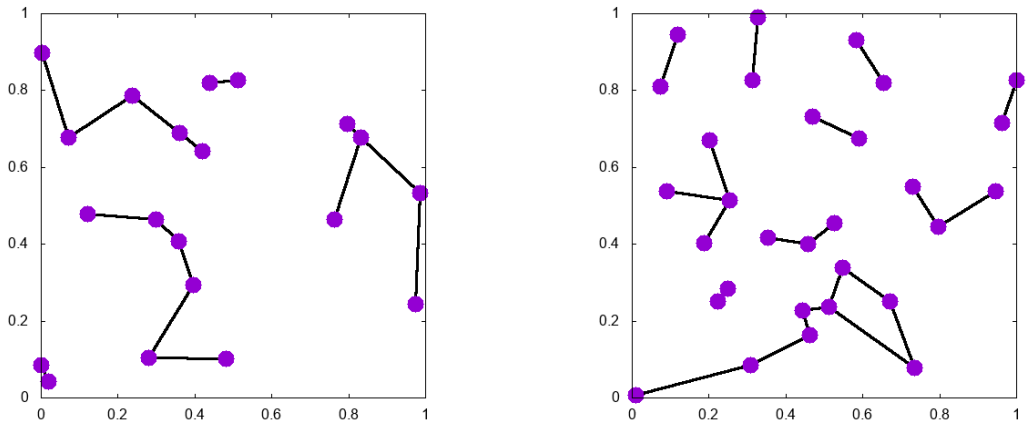
Cuadro 1: Asignación de colores en `gnuplot`.

cuenta que no todos los nodos estarían conectados entre sí, lo que provoca un grafo bosque, el cual está conformado con dos o más árboles no conexos entre sí.

En la figura 2 se pueden observar estos grafos con variaciones en la cantidad de nodos, aún cuando existen más de ellos se tiene que ninguno queda sin conectarse con otro, todos los nodos tienen al menos un nodo con el cual se conectan, sin embargo en ese caso las aristas se repiten y si se repiten el tiempo de ejecución tanto en el código como en el gráfico es mayor y podría ocasionar problemas cuando se tengan casos donde los nodos sean muchos.

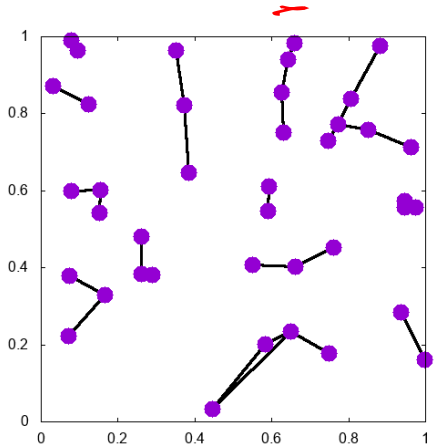
Es por eso que se hicieron modificaciones para que no existieran aristas repetidas que ayudarán a que solamente una vez se logren conectar los nodos entre sí pasando por el mismo camino. Los resultados que se obtuvieron al evaluar estos cambios están en la figura 3. Como se puede observar, no existe gran diferencia entre estos tipos de grafos y los que se tenían cuando las aristas sí estaba repetidas, lo cual nos indica que aunque se realizaron modificaciones en el código la esencia del grafo sigue siendo la misma y los resultados están presentes en esos gráficos con distinta cantidad de nodos.

Se pueden hacer más variaciones en el código ya que existen muchos tipos de grafos. En los códigos que se realizaron para la práctica se obtuvieron resultados deseados, existen diversas aplicaciones que se les pueden dar a los ciclos como por ejemplo en ruteo cuando se debe pasar una sola vez por todos los caminos visitando a todos los clientes que requieren del servicio de alguna empresa, o los grafos de bosque que nos ayudan a visualizar por ejemplo las conexiones que tienen las personas en las redes sociales.

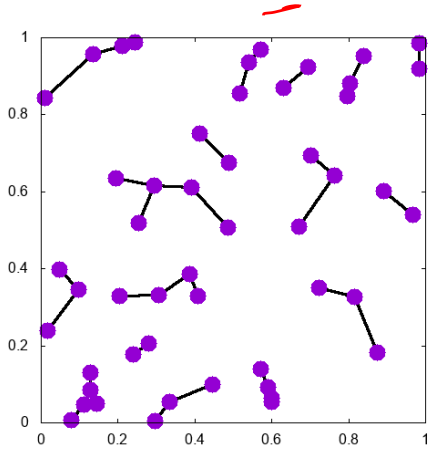


(a) 20 nodos

(b) 30 nodos

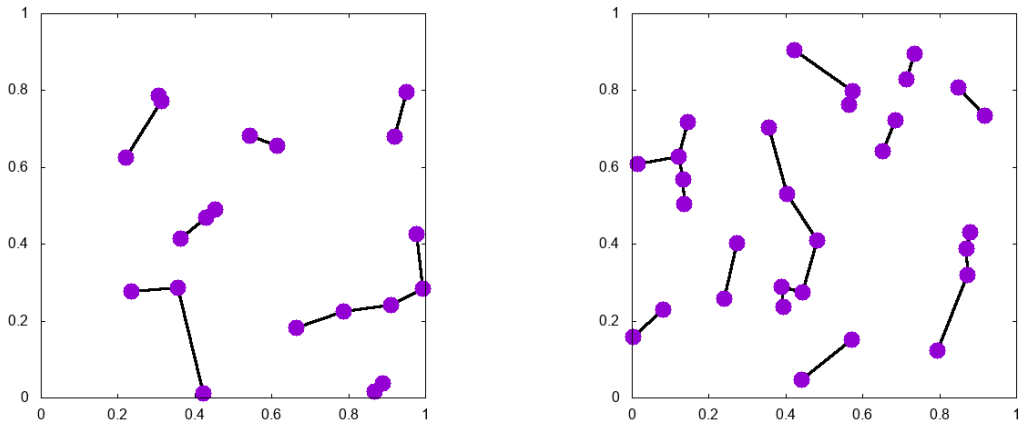


(c) 40 nodos



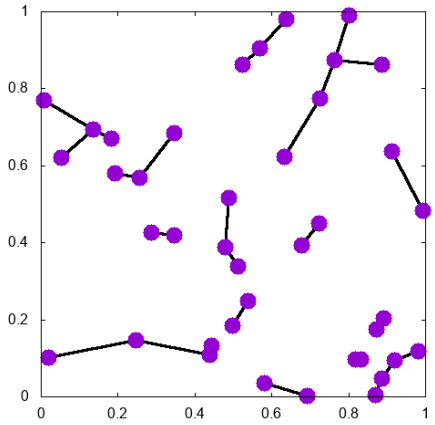
(d) 50 nodos

Figura 2: Grafos de bosque con variaciones en la cantidad de nodos.

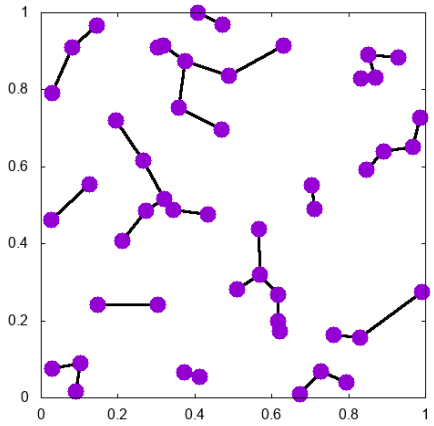


(a) 20 nodos

(b) 30 nodos



(c) 40 nodos



(d) 50 nodos

Figura 3: Grafos de bosque con variaciones en la cantidad de nodos.

Tarea 2

Se realizó una mejora en la redacción del reporte y se incluyeron referencias, se tiene una mejora en la localización correcta de los signos de puntuación tanto en los títulos como en las redacciones de las imágenes y se agregó parte del código que se realizó. Sin embargo, se tienen imágenes que no son claras de ver, por lo que se deben hacer ajustes en el código que ayuden a que la visualización sea más clara.

Tarea 2
Optimización de flujo en redes

Beatriz Alejandra García Ramos

5 de marzo de 2018

1. Introducción

En esta práctica se tiene la creación de un bosque en el que se puede decidir si se tienen direcciones y ponderaciones sobre las aristas. También en ésta se tiene que el bosque se crea a partir de la estructura de una clase en el programa `python` [1] con distintas características que deben ser señaladas en el programa `gnuplot` [2].

El diseñar el bosque a partir de una clase en `python` da la facilidad al usuario de crear tantos distintos grafos como quiera con el simple hecho de modificar los parámetros que se necesitan ingresar en el código que aportan las características del bosque, si se desea hacer dirigido o no, con ponderaciones o no y además si se desea cambiar la cantidad de nodos que tendrá el bosque.

2. Actualización del grafo en `python`

En la práctica anterior del curso de optimización de flujo en redes [3] se realiza un bosque en el cual se tienen ciertos nodos conectados entre sí por medio de aristas dependientes de un criterio, sin embargo el problema que se tiene en esa práctica es que el usuario no tiene la posibilidad de que se realicen bosques dirigidos o con ponderaciones ya que los criterios que se toman son únicos en el código realizado, lo cual ocasiona un problema si se requiere que alguna situación se visualice por medio de un grafo si éste necesita ser con dirección o ponderaciones como normalmente se utilizan en el área de flujo en redes.

Además se tiene en el código una gran cantidad de líneas que posiblemente al usuario no le interese conocer del todo, ya que el usuario se interesa en obtener el bosque final, es por eso que lo que se requiere es tener principalmente qué cantidad de nodos se requieren para el bosque y si éste debe ser dirigido, ponderado o ambos.

Es por esta razón que en esta práctica se tiene una clase que permite que el bosque se cree dependiente de los parámetros de la cantidad de nodos, direcciones y ponderaciones que el usuario indica en el llamado a las funciones definidas en la clase. Para que el bosque comience a crearse se necesita primeramente declarar los vectores, conjuntos y variables que se van a requerir en las demás funciones de la clase como se muestra a continuación.

```
class Grafo:

    def __init__(self):
        self.nodos = []
        self.aristas= []
        self.dis = []
        self.dismin = []
        self.peso = []
        self.vector = []
```

Para poder crear los nodos y conectarlos entre ellos se realizan las funciones `agrega`, `distancia` y `conecta` donde `agrega` define la coordenada de los nodos y además los va guardando en un archivo

.dat para que después se puedan graficar en gnuplot, los nodos se van tomando de manera aleatoria y una vez que se tienen todos en el vector nodos se calculan las distancias que existen entre ellos con la función distancia.

```
def agrega(self, n):
       with open("DirPesoNodos.dat", "w") as crear:
           for t in range(n):
               x=random()
               y=random()
               self.nodos.append((x,y))
               print (x,y,file=crear)

def distancia(self):
    for (x1,y1) in self.nodos:
        for (x2,y2) in self.nodos:
            d=sqrt(((y2-y1)**2)+((x2-x1)**2))
            if d==0:
                self.dis.append(20)
            else:
                if d in self.dis:
                    self.dis.append(20)
                else:
                    self.dis.append(d)
```

Una vez calculada la distancia ésta se utiliza en la función conecta la cual toma la menor distancia entre dos nodos que se encuentra en el vector dis, verifica si ésta se encuentra en el vector dismin y si no es así entonces conecta esos nodos entre sí, pero para ello se requiere saber cuál es la distancia mínima de todas las distancias calculadas, lo cual podría ser un conflicto si la distancia que se calcula entre un nodo y sí mismo es cero dado que siempre tomaría esas distancias y lo que se requiere es que cada nodo tenga al menos una conexión con otro, es por eso que la distancia entre un nodo y sí mismo en la función distancia es veinte cuando esto sucede.

```
def conecta(self):
    for (x1,y1) in self.nodos:
        self.dismin=[]
        for(x2,y2) in self.nodos:
            dm=sqrt(((y2-y1)**2)+((x2-x1)**2))
            if dm==0:
                  self.dismin.append(20)
            else:
                if dm in self.dismin:
                    self.dismin.append(20)
                else:
                    self.dismin.append(dm)
        if min(self.dismin) in self.dis:
            a1=self.nodos[self.dismin.index(min(self.dismin))][0]
            b1=self.nodos[self.dismin.index(min(self.dismin))][1]
            self.aristas.append((x1,y1,a1,b1))
            self.peso.append(ceil(random()*10))
            self.vector.append(punto((x1,y1),(a1,b1)))
            self.dis.remove(min(self.dismin))
```

Una vez que se realiza la conexión de un nodo con otro se toma de manera aleatoria una ponderación para esa arista que toma valores enteros entre un intervalo de 1 y 10 incluyéndolos, además se calcula la posición en que esta ponderación será colocada en el gráfico y se requiere para ello una función definida fuera de la clase que calcula el valor de las coordenadas de ese punto.

```
def punto(x1,y1):
    return(((x1[0] + y1[0])/2)-.005, ((x1[1] + y1[1])/2)+.005)
```

3. Resultados

Para poder graficar los datos obtenidos en `gnuplot` se requieren ciertas instrucciones que indican si el bosque será ponderado o con direcciones.

Para que un bosque sea dirigido o no se requiere de la instrucción `head` y `nohead` respectivamente, para lo cual tendremos que decidir, si la variable `di` que proporciona el usuario es cero entonces el bosque tomará la instrucción de `nohead`, si es uno tomará la instrucción de `head` que nos dará direcciones en éste.

```
if di is 1:
   print("set arrow {:d} from {:f}, {:f} to {:f}, {:f} head filled lw 1".format(num+1,x1,y1,x2
       ,y2),file=archivo)
else:
   print("set arrow {:d} from {:f}, {:f} to {:f}, {:f} nohead filled lw 1".format(num+1,x1,y1,
       x2,y2),file=archivo)
```

Entonces para indicar que el bosque debe tener dirección la variable `di` es igual a uno, lo cual nos da como resultado grafos como los de la figura 1 en donde las direcciones se toman dependiendo de qué nodo es el que se está conectando con otro.

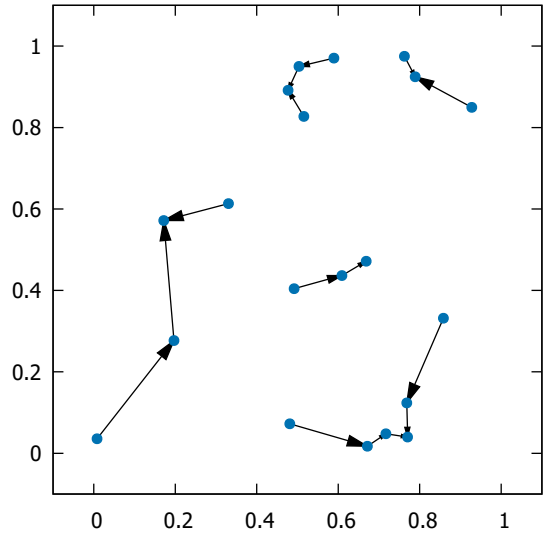
Para que un bosque sea ponderado entonces se deben tomar los datos de los vectores `peso` y `vector` que son los que indican en qué posición se pondrá la ponderación que le corresponde a cada arista. Así para la primer arista se tomaría en cuenta que $\texttt{p} = \texttt{self.peso}[0]$ y $(\texttt{kp}, \texttt{rp}) = \texttt{self.vector}[0]$ que son los datos de la primer posición de cada vector. De esta manera cada arista tendría el valor que le corresponde dado en la función `conecta`.

```
if pesos is 1:
   p = self.peso[num]
   (kp, rp) = self.vector[num]
   print("set label font ',11' '{:d}' at {:f},{:f} tc rgb 'brown'".format(p, kp, rp),file =
       archivo)
```

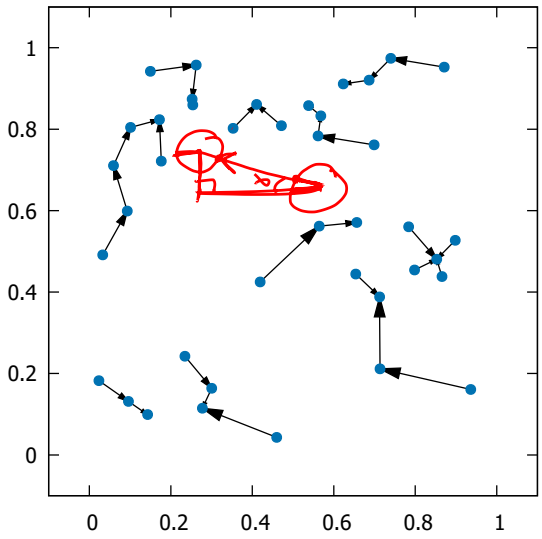
Indicando que la variable `pesos` es igual a uno tendremos un bosque ponderado como los que se encuentran en la figura 2 que varían en nodos y tiene ponderaciones determinadas en un intervalo entre uno y diez tomando los enteros como se puede observar en los valores indicados en cada arista, donde `i` es la cantidad de nodos del bosque.

De la misma manera se tendrá que cuando ambas variables sean iguales a uno el bosque será dirigido y ponderado de manera aleatoria como se observa en la figura 3.

```
i=40
G = Grafo()
G.agrega(i)
G.distancia()
G.conecta()
G.graficar(di=1, pesos=1)
```

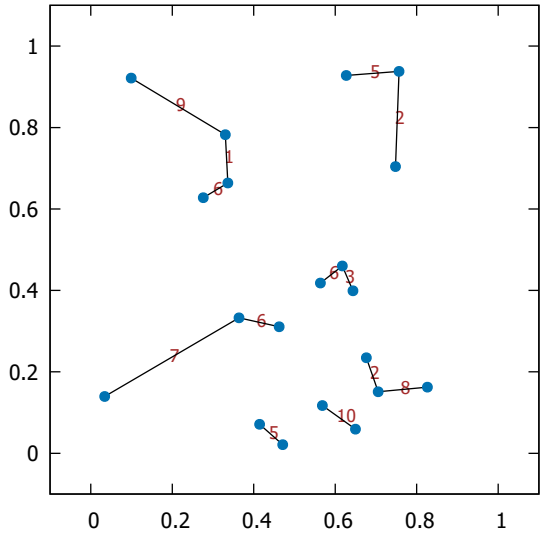


(a) 20 nodos.

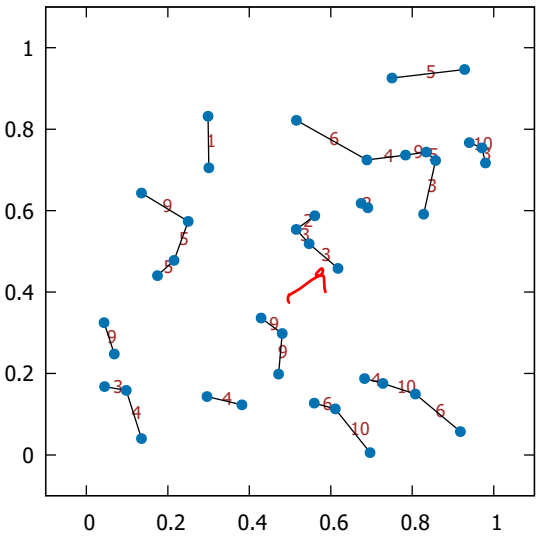


(b) 40 nodos.

Figura 1: Grafos de bosque dirigidos.

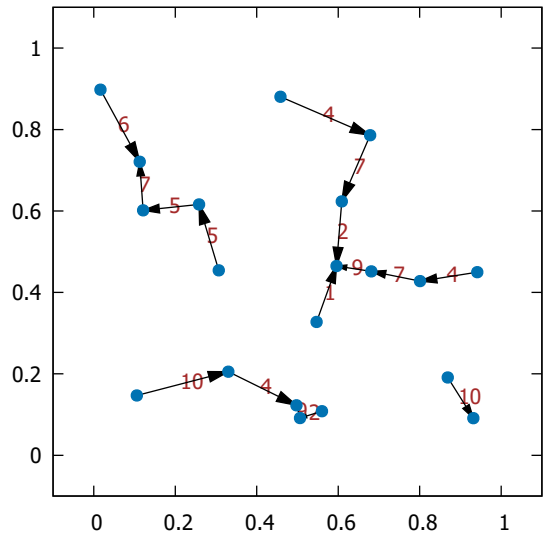


(a) 20 nodos.

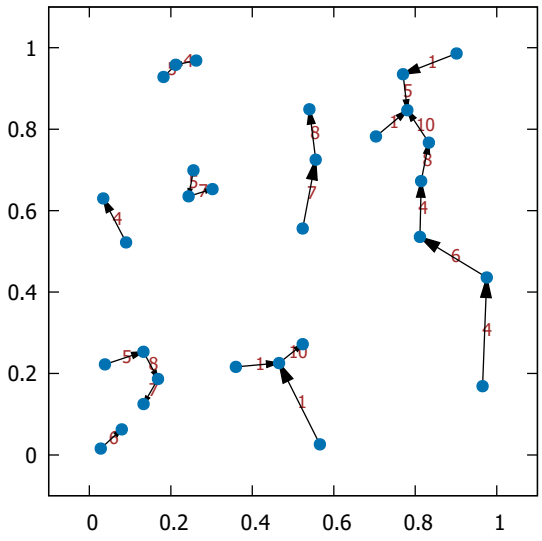


(b) 40 nodos.

Figura 2: Grafos de bosque con ponderaciones.



(a) 20 nodos.



(b) 30 nodos.

Figura 3: Grafos de bosque dirigidos y con ponderaciones.

Referencias

[1] Python Software Foundation [US]. (2001-2018). https://www.python.org/downloads/

[2] Gnuplot. http://www.gnuplot.info/

[3] Beatriz García. Tarea 1. Optimización de flujo en redes. https://github.com/BeatrizGarciaR/FlujoEnRedes/blob/master/Tarea1/Tarea1.pdf, 2018.

Tarea 3

Se tienen problemas de redacción y del tipo de formato que se le debe dar a algunos conceptos que se utilizan en el reporte. Se realizó la mejora de las imágenes del grafo realizado, pero no se tiene un buen ajuste de curva en las gráficas de los tiempos de ejecución y las referencias que se utilizaron no son adecuadas para un reporte científico.

Tarea 3
Optimización de flujo en redes

Beatriz Alejandra García Ramos

9 de abril de 2018

1. Introducción

En esta práctica se realizaron modificaciones en el código hecho para la segunda práctica [1] en donde se crea un grafo, ya sea dirigido y ponderado, solo uno de ellos o ninguno de los dos. Estas modificaciones fueron incluir dos algoritmos, el de Ford-Fulkerson [2] y el de Floyd-Warshall [3], los cuales nos ayudan a encontrar el flujo máximo de un nodo inicio a un nodo fin y el camino más corto de un nodo a otro, respectivamente.

Se obtuvieron resultados de los algoritmos y además se realizó un análisis de los tiempos de ejecución con variante en la cantidad de nodos del grafo, éstos se graficaron en cajas de bigotes utilizadas en estadística para visualizar la distribución del conjunto de datos y se realizó una comparación entre la manera en que los tiempos se comportan y la función exponencial.

2. Modificación del grafo en `python`

Dado que los algoritmos que se incluyeron en el código de `python` están de acuerdo a nodos y aristas con propiedad de conjunto y diccionario respectivamente, se hizo la modificación del código implementado para la segunda práctica llamada `Tarea2` de modo que los nodos fueran guardados en una variable `set`() para que no se incluyan dos nodos iguales y al conectarse las aristas estuvieran en una variable `dict`() que permite etiquetar a cada una de las aristas con la capacidad correspondiente a la conexión de un nodo con otro.

Dado que se realizaron modificaciones en el tipo de las variables se realizaron también los cambios necesarios al criterio de conexión de nodos y a las instrucciones que se le daban a `gnuplot` para realizar el grafo correspondiente. Además se creó una función en donde se guardan los nodos correspondientes al grafo, así pueden identificarse el nodo inicio, indicado con naranja, y el nodo fin, indicado con rojo, que se utilizan para el algoritmo de Ford-Fulkerson.

Anteriormente la visualización del grafo era poco apropiada, así que se realizaron modificaciones para que el grafo que se crea tenga colores cuando existen capacidades en las aristas, estos colores están dados por intervalos como se muestran en el cuadro 1, para las capacidades que van entre uno y tres se tiene el color morado en las aristas, cuando se tiene capacidad en la arista ente cuatro y seis entonces ésta se colorea de verde y si se tienen aristas de color azul sus capacidades pueden ir entre siete y diez.

Cuadro 1: Asignación de colores para las capacidades de las aristas.

Color	Conjuntos
Morado	{1, 2, 3}
Verde	{4, 5, 6}
Azul	{7, 8, 9, 10}

3. Inclusión de los algoritmos de Floyd-Warshall y Ford-Fulkerson

El algoritmo de Floyd-Warshall busca todos los caminos posibles de todos los nodos hacia todos los demás. En la práctica anterior se tenía un bosque, sin embargo, para que existieran más posibilidades de encontrar caminos de un nodo a otro se crearon aristas aleatorias que conectan nodos con otros aunque esto ya no forme un bosque, así cuando se tiene el algoritmo de Floyd-Warshall se toman en cuenta los caminos posibles de un nodo a otro y se guarda el de menos distancia.

En el caso en que se tiene el algoritmo de Ford-Fulkerson se toman en cuenta todas las capacidades de las aristas desde el nodo inicio hasta el nodo fin y de acuerdo al proceso de éste se calcula la mayor cantidad de flujo que puede enviarse sin importar qué camino se tome para llegar de un nodo a otro.

Para poder lograr que los algoritmos de Floyd-Warshall y de Ford-Fulkerson funcionen en cualquier tipo de grafo, ya sea dirigido, con pesos, con alguno de éstos, o con ninguno, se realizaron modificaciones en el código y una de las importantes para los algoritmos es si el grafo es dirigido o no, cuando es dirigido solo se toma el camino del primer nodo seleccionado al segundo nodo, sin embargo, cuando no es dirigido la dirección se toma del primer nodo al segundo nodo y del segundo al primero y la capacidad que se les asigna es la misma.

En la figura 1 se tienen grafos sin dirección, puede visualizarse que cuando los grafos tienen capacidades se involucran los colores en las aristas, cuando el grafo no es dirigido el algoritmo de Floyd-Warshall y el de Ford-Fulkerson realizan su trabajo, pero si se tiene que la capacidad de las aristas es cero entonces ambos obtienen resultados de cero en distancias y cero en flujo máximo. Por otro lado, cuando las aristas tienen capacidades, se tiene que el algoritmo de Floyd-Warshall obtiene la distancia mínima de un nodo a otro tomando en cuenta distintos caminos para llegar a él y el algoritmo de Ford-Fulkerson obtiene el flujo máximo entre el nodo inicio y el nodo fin y devuelve el valor de ese flujo, si es que existen aristas entre el nodo inicio y el nodo fin.

En el caso en que sí existe una dirección, como se muestra en la figura 2, y que además existen capacidades en las aristas se tiene que el algoritmo de Ford-Fulkerson tiene menos posibilidad de obtener un resultado, ya que los nodos inicio y fin podrían no tener acceso entre ellos de acuerdo a las direcciones que se tomaron aunque existan aristas entre ellos, de la misma manera afecta al algoritmo de Floyd-Warshall ya que se tiene que no todos los nodos pueden llegar a los demás. Cuando no hay capacidades entonces todos los resultados son iguales a cero.

4. Tiempos de ejecución de los algoritmos

Para el caso en que no se tienen capacidades los algoritmos tienen como resultado cero, sin importar si existe dirección o no.

Cuando se tiene que sí existen capacidades en las aristas mayores que cero entonces se tiene que el algoritmo de Floyd-Warshall encuentra una distancia mínima entre un nodo y otro siempre y cuando éstos estén conectados por aristas, pero cuando se tiene que existen conexiones pero con direcciones es posible que para algún nodo no existan entradas, solo salidas y ésto complica el resultado del algoritmo, aunque existan aristas, si no hay entradas a algún nodo entonces la distancia será cero y es por eso que en la teoría se tiene que la complejidad computacional del algoritmo de Floyd-Warshall es $O(V^3)$ donde V es el número de nodos del grafo.

En el caso en que se tiene que sí existen capacidades para las aristas y no hay dirección el algoritmo de Ford-Fulkerson no tiene ningún problema en encontrar el flujo máximo cuando existen conexiones entre el nodo inicio y el nodo fin ya que se puede llegar de uno a otro en ambos sentidos, pero cuando existen direcciones se tiene que es probable que no haya una entrada al nodo fin desde el nodo inicio aunque haya aristas conectadas entre ellos, es por eso que la posibilidad de que exista un resultado de este algoritmo es mayor cuando no hay direcciones, es por eso que la complejidad computacional está dada por $O(E|f^*|)$ donde f es el máximo flujo y E es la cantidad de aristas en el grafo.

De acuerdo al funcionamiento y la complejidad computacional de cada uno de los algoritmos se tiene que se analizó el tiempo de ejecución del código en `python`, los tiempos fueron guardados en archivos `.csv` que después fueron importados al programa `R` para lograr hacer cajas de bigote que nos ayudan a visualizar el comportamiento de los tiempos de ejecución cuando se realizan repeticiones para

cantidades de nodos distintas. Además se verificó que las repeticiones para cada una de las cantidades de nodos no se distribuyen normalmente, es por eso que las cajas de bigote varían en su proporción.

De esta manera, como se puede ver en la figura 3 se tienen las repeticiones de las cantidades de nodos variantes cuando se hacen corridas de grafos que no tienen dirección. Cuando se tienen grafos con direcciones se obtuvieron los datos que se muestran en la figura 4.

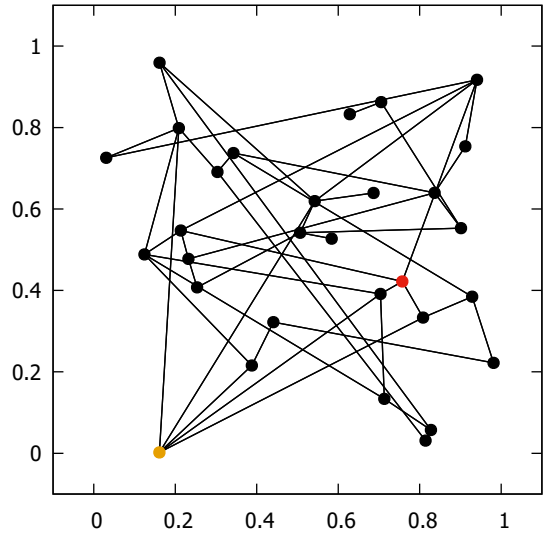
Es claro que cuando se tienen grafos sin dirección el tiempo de ejecución aumenta, ya que existen más posibilidades de encontrar caminos entre un nodo y otro, por lo cual la complejidad de ambos algoritmos aumenta y hace que el tiempo de ejecución aumente también. Aún así los tiempos de ejecución tanto cuando hay dirección como cuando no la hay no tienen gran diferencia entre ellos y en ambos casos se comportan de manera exponencial, como se muestra en la curva azul de las figuras, y este comportamiento es común cuando se aumentan los nodos ya que las aristas también aumentan y es un trabajo mayor para el sistema computacional, además se ve afectado por las demás aplicaciones que se utilizan en el momento en que se están ejecutando los grafos.

5. Conclusiones

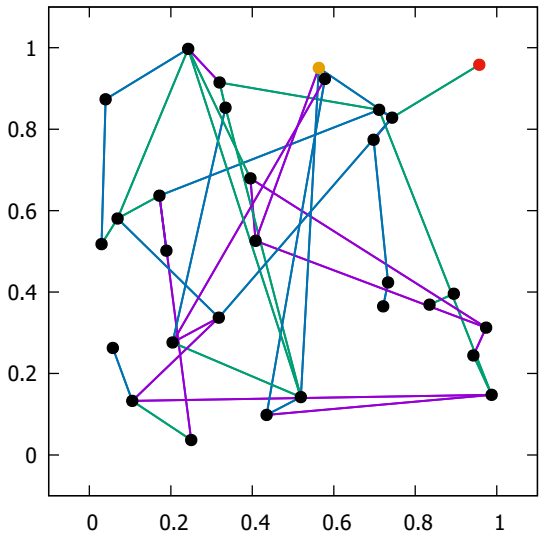
Las modificaciones que se hicieron en el grafo son de gran ayuda para que no existan tantas instrucciones dentro del código y se tenga una mejora en el tiempo de ejecución de los grafos. Además adaptando los algoritmos de Ford-Fulkerson y Floyd-Warshall no se les hacen modificaciones a éstos por lo que se puede asegurar que funcionan de manera correcta.

Cuando no se tienen capacidades en las aristas es claro que los algoritmos no pueden arrojar una respuesta distinta a cero y eso no es en lo que se enfoca el estudio de éstos, es por eso que siempre que se utilicen debemos asegurarnos de que sí existan capacidades en las aristas para que cuando se tenga un grafo dirigido o no se obtengan buenos resultados, aunque éstos sean cero en algunos casos por el hecho de que no exista conexiones entre los nodos con algunos otros.

Los tiempos de ejecución aumentan de manera significativa entre una cantidad de nodos y otra, en esta práctica se tienen variaciones de cinco nodos entre un análisis de tiempo y otro, y aún así logra verse que los tiempos aumentan de acuerdo a su tamaño y cada vez son más complejos de acuerdo a la complejidad computacional de ambos algoritmos.

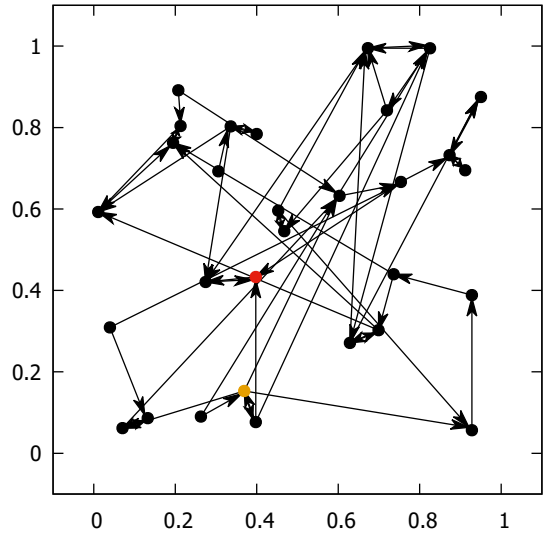


(a) Sin capacidad.

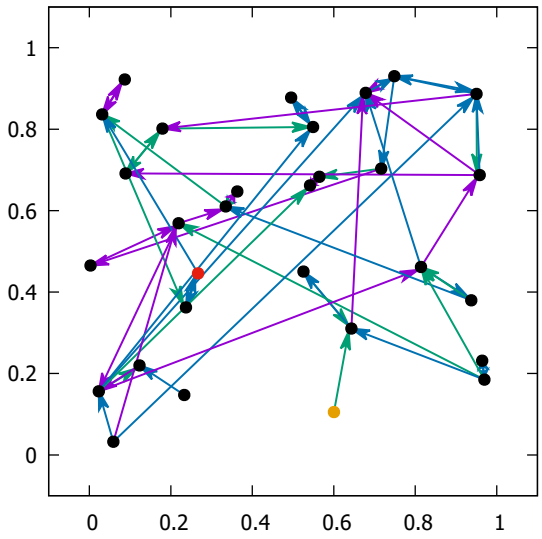


(b) Con capacidad.

Figura 1: Grafos sin dirección.



(a) Sin capacidad.



(b) Con capacidad.

Figura 2: Grafos con dirección.

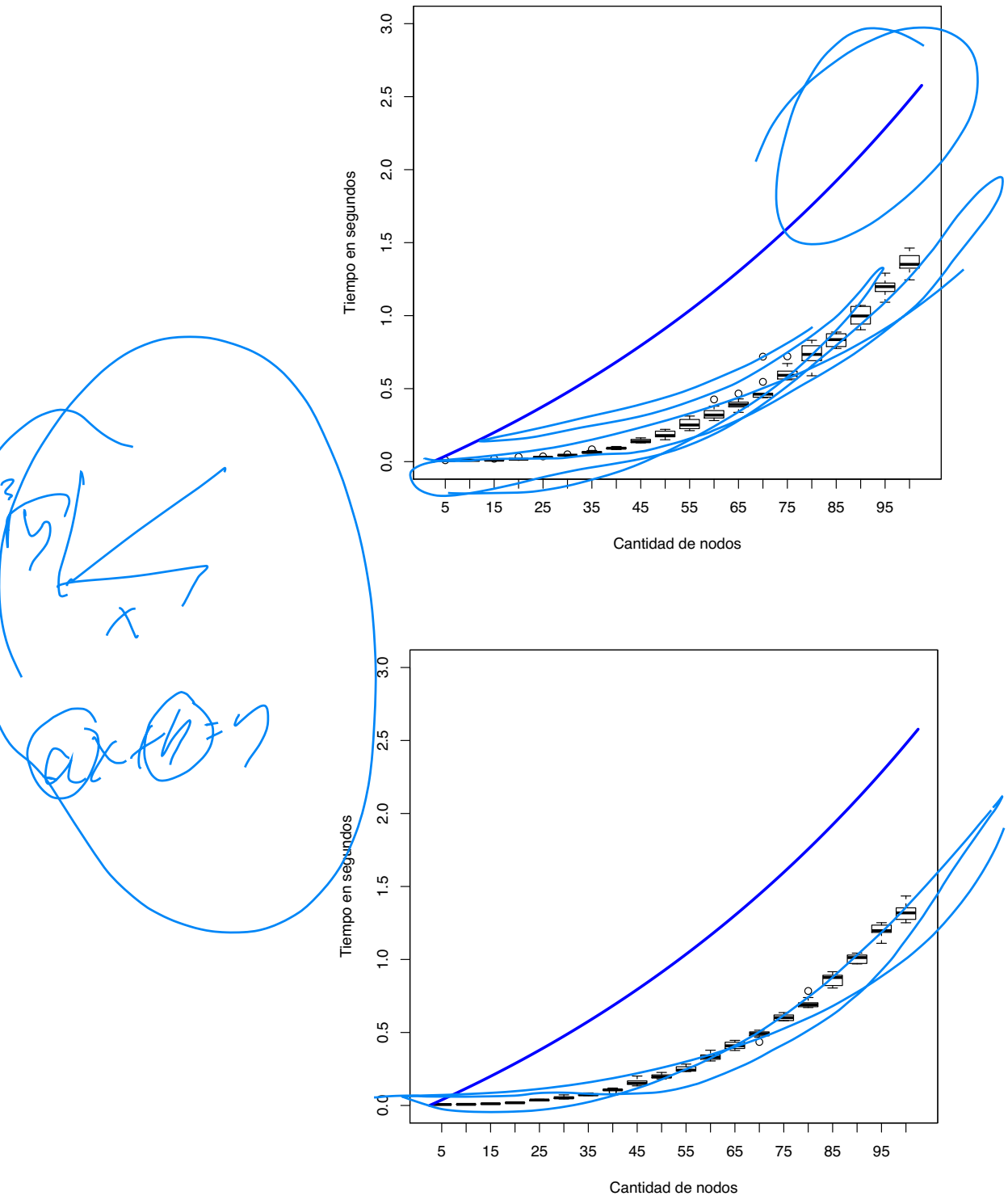
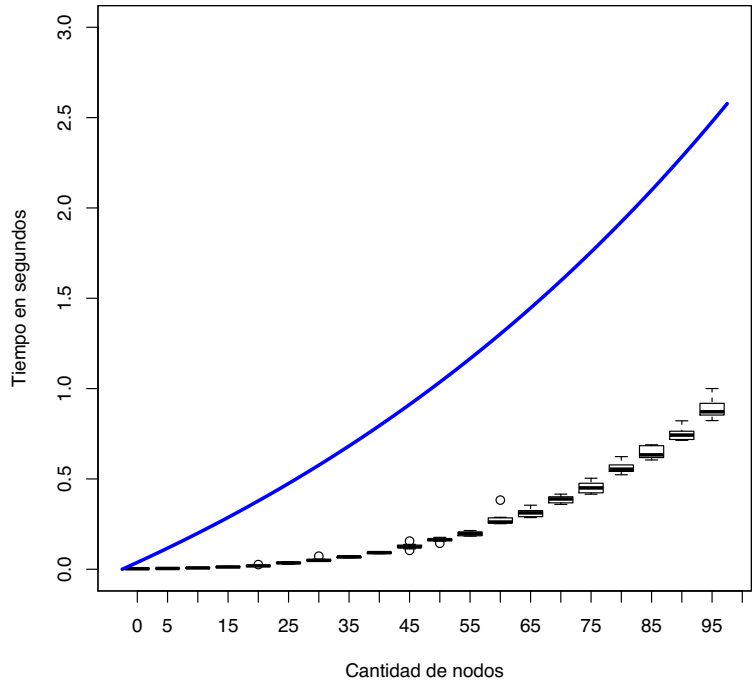


Figura 3: Grafos sin dirección.



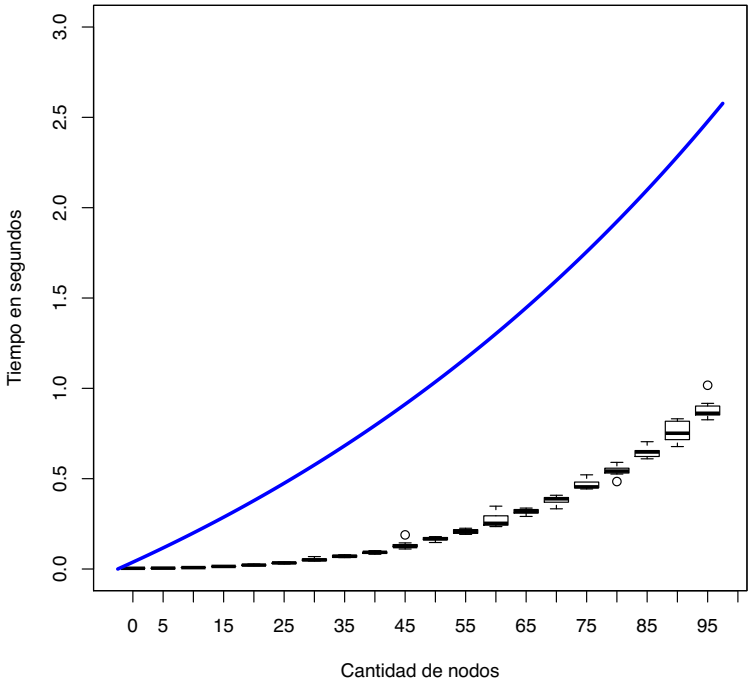


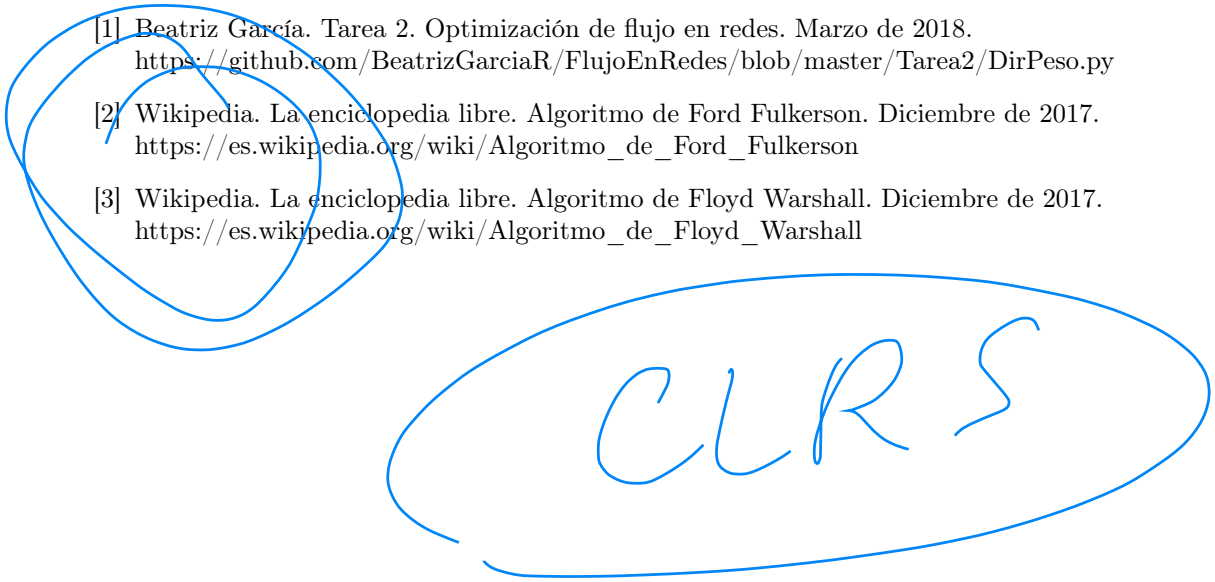
Figura 4: Grafos con dirección.

Referencias

[1] Beatriz García. Tarea 2. Optimización de flujo en redes. Marzo de 2018. https://github.com/BeatrizGarciaR/FlujoEnRedes/blob/master/Tarea2/DirPeso.py

[2] Wikipedia. La enciclopedia libre. Algoritmo de Ford Fulkerson. Diciembre de 2017. https://es.wikipedia.org/wiki/Algoritmo_de_Ford_Fulkerson

[3] Wikipedia. La enciclopedia libre. Algoritmo de Floyd Warshall. Diciembre de 2017. https://es.wikipedia.org/wiki/Algoritmo_de_Floyd_Warshall



Tarea 4

Se tienen problemas de redacción y puntuación de nuevo y además se tienen problemas en los ejes que son colocados en las gráficas de las imágenes del grafo y además en los ejes marcados en los criterios de distancia y densidades que se tomaron de las variaciones de los parámetros dados.

Tarea 4
Optimización de flujo en redes

Beatriz Alejandra García Ramos

23 de abril de 2018

1. Introducción

En esta práctica se considera la ejecución del algoritmo de Floyd-Warshall antes implementado en un grafo, como se trabajó en la Práctica 3 [1]. Para analizar este algoritmo se considera calcular el promedio de las distancias y la densidad en el grafo. Además se evalúa el tiempo de ejecución haciendo variaciones en la cantidad de nodos.

2. Creación del grafo

Anteriormente se había trabajado con un grafo en donde había cierto criterio de conexión entre los nodos, la cual los conectaba de manera que tomara él que estuviera más cercano a él y se creaba un bosque.

Para esta práctica se tiene que el grafo creado toma en cuenta la posición de los nodos de acuerdo a una circunferencia. Los nodos son colocados al calcularse el ángulo que le corresponde a partir de la cantidad de nodos que se dan, así si se tienen, por ejemplo, cincuenta nodos, cada uno estará colocado a siete grados con dos minutos aproximadamente del otro, como se muestra en la figura .

Una vez colocados los nodos ahora se establecen conexiones entre ellos, los cuales se llevan a cabo con un criterio `k` de distancia, así cuando se va variando este criterio a partir de $k = 1$ se tiene que los nodos están conectados solamente con el nodo siguiente y el anterior a él, cuando $k = 2$ se tiene entonces que el nodo se conecta con dos anteriores a él y con dos siguientes, y así sucesivamente.

En la figura 1 se puede observar un grafo con cincuenta nodos en donde el criterio de conexión `k` es igual a trece. De esta manera se tendrían distintos caminos para llegar de un nodo a otro y el algoritmo de Floyd-Warshall tendría que encontrar el más corto de ellos. El criterio de conexión puede variar tanto como la mitad de los nodos existentes, ya que si se quisieran tomar más de esa cantidad se repetirían las conexiones.

3. Distancia promedio y densidad promedio

Una vez que se realiza el algoritmo de Floyd-Warshall se calculan las distancias de un nodo hacia los demás para todos los nodos en el grafo si existe un camino que los conecte, los resultados son guardados en una lista y ésta es utilizada para calcular la distancia promedio. Sin embargo, la distancia promedio resultante es un número mayor que uno.

Para poder tener una comparación con las densidades, las distancias promedios se deben normalizar, para esto creamos una cota, la cual nos ayudará a dividir las distancias promedios para que los resultados obtenidos estén en un intervalo entre cero y uno y las curvas puedas ser comparadas.

Las distancias promedio se calculan como normalmente se calcula un promedio, se toman todas las distancias que son generadas en el algoritmo de Floyd-Warshall, se suman y el total de la suma se divide entre la cantidad de distancias calculadas en el algoritmo.

Para calcular las densidades del grafo lo que se desea hacer es tomar cada nodo del grafo, para cada uno de ellos ver los nodos vecinos correspondientes, es decir, los nodos con los cuales tienen conexión,

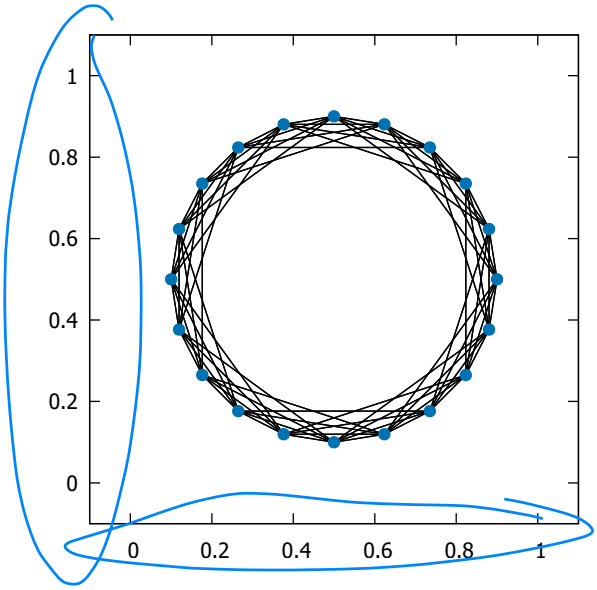


Figura 1: Grafo creado con cincuenta nodos.

una vez identificados los vecinos se requieren contabilizar las conexiones que tienen esos vecinos. Una vez que se calcula la cantidad de conexiones que hay entre los vecinos ésta se divide entre la cantidad de conexiones que pueden llegar a existir entre los vecinos, así se hace para cada nodo en el grafo y el resultado es la densidad de cada uno de ellos.

Para obtener la densidad del grafo se requiere sacar el promedio de las densidades de cada nodo, así, de manera similar al calcular el promedio de las distancias, se calcula el promedio de las densidades y se obtiene como resultado un valor entre cero y uno.

Ya que se tienen funciones para calcular ambos promedios se realizan variaciones en el criterio `k` con diez probabilidades de conexión aleatoria distintas y se guardan los datos en archivos `.csv`, los cuales son trabajados en el programa `R`.

Como se puede ver en la figura 2 se tiene como resultados dos curvas, la que se encuentra marcada con color azul es la que muestra las densidades promedio obtenidas, la que se encuentra de color rojo es la que muestra las distancias promedio normalizadas dada la cota. En la figura se tiene que las distancias se comportan de manera casi constante, esto sucede porque para las distancias en el algoritmo de Floyd-Warshall, aún cuando se varía el criterio de conexión, se buscan las distancias más cortas y el promedio de éstas es similar.

4. Tiempos de ejecución

Para poder analizar los tiempos de ejecución se requiere hacer variaciones en la cantidad de nodos que tiene el grafo, además se realizan repeticiones para cada tamaño, de manera que se tiene una probabilidad menor de error al calcular el tiempo real, ya que si el procesador está variando su funcionamiento con distintas aplicaciones que se encuentran activas puede haber cierta diferencia entre una proceso y otro aún cuando se tenga exactamente el mismo número de nodos en el grafo.

Como se observa en la figura 3 los tiempos de ejecución aumentan de tal manera que aumenta la cantidad de nodos del grafo. Esto se debe a que la complejidad computacional del algoritmo de Floyd-Warshall depende del número de nodos, de manera que la complejidad es de n^3. Por lo tanto, se tiene que la curva que se genera en los tiempos de ejecución se comporta como una curva cúbica.

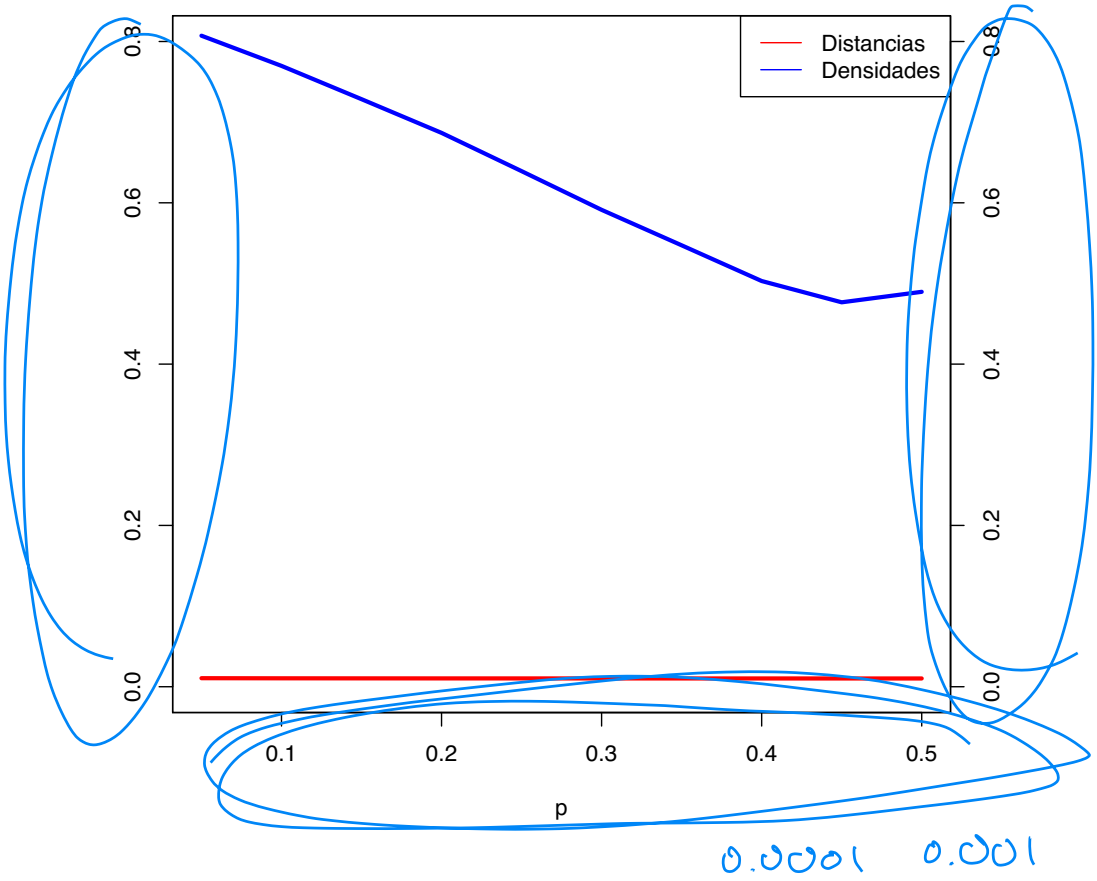


Figura 2: Variaciones en el criterio `k` y la probabilidad de conexión `p` con un grafo de cien nodos.

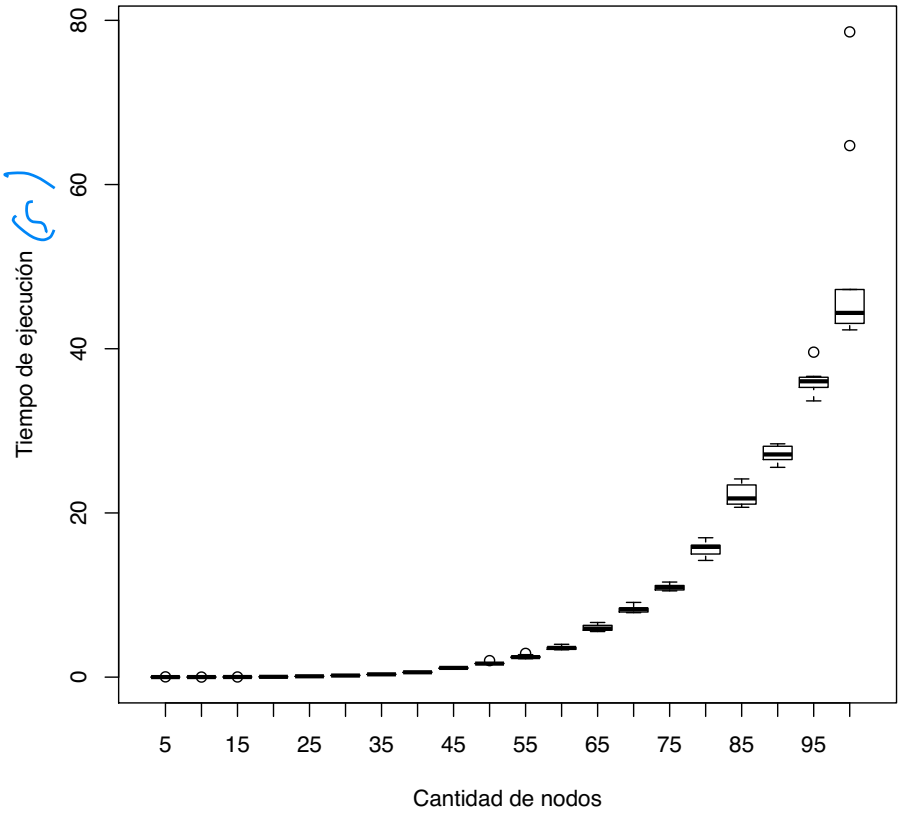


Figura 3: Tiempos de ejecución con variaciones en cantidad de nodos.

Referencias

[1] Beatriz García. Tarea 3. Optimización de flujo en redes. Abril de 2018. https://github.com/BeatrizGarciaR/FlujoEnRedes/blob/master/Tarea3/Tarea3Completo.py

Tarea 5

Para esta práctica se tienen problemas de redacción en algunas ocasiones y además se buscaban eliminar los ejes en las gráficas que se realizaron para los grafos pero no se logró eliminar el cuadro que se genera al encerrar el plano cartesiano y eso hace que las imágenes no sean adecuadas para un trabajo científico, además faltan especificaciones sobre lo que significan los ejes coordenados en las gráficas que analizan el flujo máximo en cada variación en los parámetros del grafo.

Tarea 5
Optimización de flujo en redes

Beatriz Alejandra García Ramos

7 de mayo de 2018

1. Introducción

En esta práctica se tiene un análisis del algoritmo de Ford-Fulkerson que ya se ha trabajado en la tercer práctica [1]. Para realizar el análisis se tienen mallas en donde se realizan percolaciones [2] de nodos o de aristas y se lleva a cabo el algoritmo de Ford-Fulkerson para verificar si existe diferencia o no en el flujo máximo que se envía desde el nodo inicial hasta el nodo final.

Además de analizar el cambio en la cantidad de flujo que se tiene al percolar nodos o aristas se tiene también el estudio en el tiempo de ejecución al realizar estos cambios realizando variaciones en el tamaño del umbral [3] que define las conexiones de los nodos a sus vecinos dependiendo del umbral que se indique.

2. Creación del grafo en `python`

Para realizar una malla de las que se requieren en esta práctica se tiene un plano donde los ejes están dados por números enteros positivos. La malla es de tamaño $n \times n$, donde n es un número dado por el usuario y $n \times n$ representa la cantidad total de nodos que habrá en esa malla.

Una vez definida la cantidad total de nodos para esa malla se tiene que el primer nodo que se crea es el que tiene como coordenada $(1, 1)$ y el último en crearse es el que tiene como coordenada (n, n), además los nodos intermedios se van creando hacia la derecha y hacia arriba.

Ya que los nodos son creados se realizan las conexiones dependiendo del umbral deseado. El umbral definirá cuáles serán los vecinos de los nodos teniendo en cuenta la distancia manhattan [4]. Si el umbral es igual a uno entonces los vecinos del nodo serán aquellos que están a una unidad en distancia manhattan, cuando se tenga umbral de dos los vecinos serán aquellos que estén a dos unidades en distancia manhattan, y así sucesivamente. De esta manera se realizan conexiones bidireccionales con capacidades que siguen una distribución normal con media igual a cuatro.

Ya que se tienen todas las conexiones dependientes del umbral se crean aristas donde un nodo se conecta con otro de manera aleatoria con una probabilidad de uno en cien, estas conexiones tienen dirección y tienen capacidades que siguen una distribución exponencial. Para que sean creadas correctamente se verifica que la conexión no sea alguna de las que ya existen al tomar en cuenta el umbral.

3. Percolación de nodos y conexiones

Cuando se tiene una malla existen diversos caminos que se pueden tomar desde el nodo inicial hasta el nodo final, lo cual ayuda a que se tenga un buen flujo al momento de realizar el algoritmo de Ford-Fulkerson.

Supongamos que la malla representa las tuberías que se tienen en una colonia, las cuales transportan el agua que llega a los hogares de las familias que viven ahí. Lo ideal sería que todas las tuberías funcionen a la perfección, sin embargo, en la vida real eso no sucede, existe la posibilidad de que

alguna tubería no funcione, que alguna de las conexiones entre una tubería y otra esté tapada, o ambas al mismo tiempo.

Como las situaciones mencionadas pueden llegar a suceder no podemos tener una malla perfecta para analizar el flujo máximo con el algoritmo de Ford-Fulkerson así que se realiza una percolación de nodos, los cuales representan las conexiones entre las tuberías, y las conexiones, que representan la tubería. Una vez realizados los cambios en las mallas se vuelve a utilizar el algoritmo y se nota una diferencia en el flujo transferido entre el nodo inicial y el nodo final.

3.1. Percolación de nodos

Para realizar percolación en nodos se tiene que crear una lista en donde se guarden aquellos nodos que se eliminan de los nodos originales, cada vez que se agrega a la lista el nodo percolado se recorre el diccionario que contiene las conexiones entre el nodo y sus vecinos y se eliminan las conexiones que existían, ya que esos caminos ya no están disponibles para el algoritmo de Ford-Fulkerson. Si las conexiones son bidireccionales se eliminan ambas direcciones, si es unidireccional se elimina la dirección, como se muestra en la siguiente parte del código.

```
        for n in self.auxNodo:
            for m in self.nodos:
                if (n,m) in self.aristas:
                    del self.aristas[(n,m)]
                if (m,n) in self.aristas:
                    del self.aristas[(m,n)]
```

En la figura 1 se puede observar cómo en una malla de tamaño 8×8 con un umbral igual a uno se realiza la percolación de algunos nodos y cómo se desvanecen también las aristas que acompañan a dichos nodos.

En la figura existe aún camino para llegar del nodo inicial, marcado en color naranja, y el nodo final, marcado en color rojo, por lo que se tiene aún un flujo positivo al momento de realizar el algoritmo de Ford-Fulkerson. Puede suceder, como se muestra, que exista algún nodo que no fue percolado pero que no tiene conexiones con otros nodos ya que el umbral no permite que logre conectarse con los nodos disponibles. También puede suceder que ya no exista camino entre el nodo inicial y el nodo final.

Para analizar el cambio que se genera en el algoritmo de Ford-Fulkerson al momento de percolar nodos se tienen variaciones en el tamaño del umbral y además se tienen iteraciones en donde cada vez que ocurre una iteración se va percolando un nodo.

Para lograr ver una diferencia entre un flujo y otro se realizan repeticiones al quitar cada nodo y se tienen cajas de bigote que incluyen las repeticiones obtenidas.

Como se puede ver en la figura 2, se tienen diferencias en las repeticiones aunque se haya percolado la misma cantidad de nodos, esto sucede por el hecho de que el nodo eliminado no es el mismo por cada repetición y las capacidades de las conexiones puede variar de tal manera que aumente o disminuya el flujo máximo, sin embargo, se tiene una visualización de la disminución de flujo al ir percolando los nodos, por lo que se tiene un efecto descendiente cada vez que se van eliminando nodos y conexiones en la malla y es muy notable la diferencia de flujos entre un umbral y otro.

3.2. Percolación de conexiones

Para realizar la percolación de conexiones se tiene que la eliminación se va realizando de veinte conexiones por iteración, dado que el número de conexiones es muy elevada. Cuando se eliminan las conexiones no se deben eliminar los nodos involucrados ya que los nodos pueden tener conexiones con algunos otros vecinos. De la misma manera que en la percolación de nodos se debe tener una lista en la cual se agregan las conexiones que se han eliminado para que no existan repeticiones de eliminación ya que no se puede eliminar lo ya eliminado, esto se realiza como se tiene a continuación.

```
    if arista not in self.lqq:
            self.lqq.append(arista)
            del self.aristas[arista]
```

En la figura 3 se muestra cómo se tiene una malla con percolaciones de aristas y en este caso ya no se encuentra un camino para llegar del nodo inicial al nodo final por lo que no se tiene un flujo al momento de realizar el algoritmo de Ford-Fulkerson.

Para poder analizar la percolación de conexiones se tiene que se eliminan veinte conexiones diez veces, se varía el umbral para ver la diferencia entre flujos de un umbral pequeño a uno más grande y se hacen repeticiones para cada eliminación de veinte conexiones.

Como se muestra en la figura 4 el flujo máximo va aumentando a medida que se incrementa el umbral, pero cada vez que se realizan percolaciones de conexiones se va perdiendo flujo a través de las conexiones restantes, en total se realiza percolación de doscientas conexiones, pero en comparación con la percolación de nodos, al percolar conexiones no se tiene una disminución drástica entre un flujo máximo y otro.

4. Análisis en el tiempo de ejecución

Al momento de realizar las percolaciones de nodos y conexiones se lleva a cabo un tiempo de ejecución mayor que si se tiene solamente la malla original ya que el algoritmo de Ford-Fulkerson es llevado a cabo cada vez que se percola un nodo o que se percolan conexiones.

Como se muestra en la figura 5, se tiene el tiempo de ejecución en cajas bigote cuando se realizan diez repeticiones para realizar percolaciones de nodos al realizar variaciones en el tamaño del umbral. Conforme aumenta el tamaño del umbral aumenta también el tiempo de ejecución dado que existen muchas más aristas entre un tamaño de umbral y otro.

En la figura 6 se tienen las mismas variaciones de umbral y se calcula el tiempo de ejecución en segundos de la percolación de conexiones con sus repeticiones. Es notable que el tiempo de ejecución es mejor que en la percolación de nodos, esto se debe a que en la percolación de nodos se realizan muchas más iteraciones que en la percolación de conexiones. Sin embargo, se puede observar que ambos tiempos de ejecución tienen el mismo comportamiento al aumentar el umbral. Mientras más conexiones se tengan más tiempo tardará en procesarse el algoritmo de Ford-Fulkerson, como ya se había mencionado en la tercer práctica.

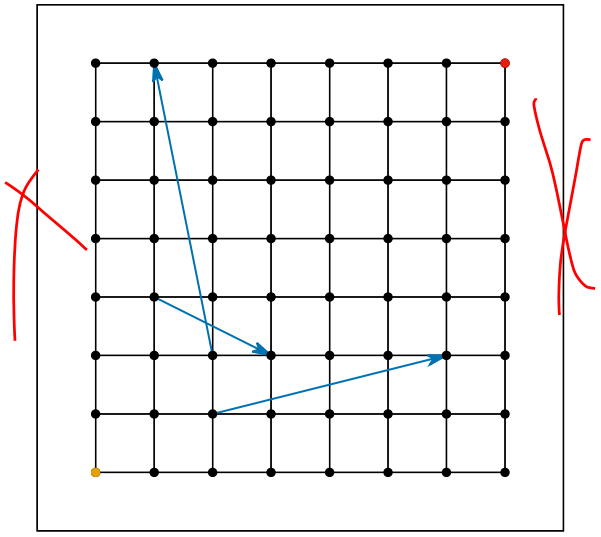
Referencias

[1] Beatriz García. Tarea 3. Optimización de flujo en redes. Abril de 2018.
https://github.com/BeatrizGarciaR/FlujoEnRedes/tree/master/Tarea3

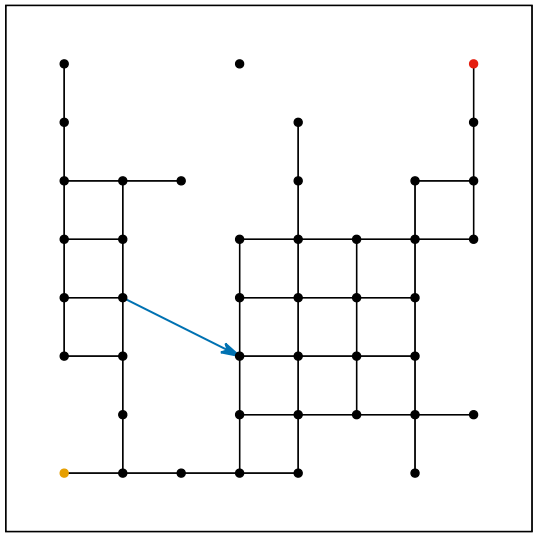
[2] Percolación en el plano con simulaciones. Laura Esvala. Mayo de 2010.
http://tikhonov.fciencias.unam.mx/presentaciones/2010may27.pdf

[3] Significado de umbral. Octubre de 2013.
https://www.significados.com/umbral/

[4] Las distancias en Manhattan. Septiembre de 2017.
https://metode.es/revistas-metode/secciones/cajon-de-ciencia/les-distancies-a-manhattan.html

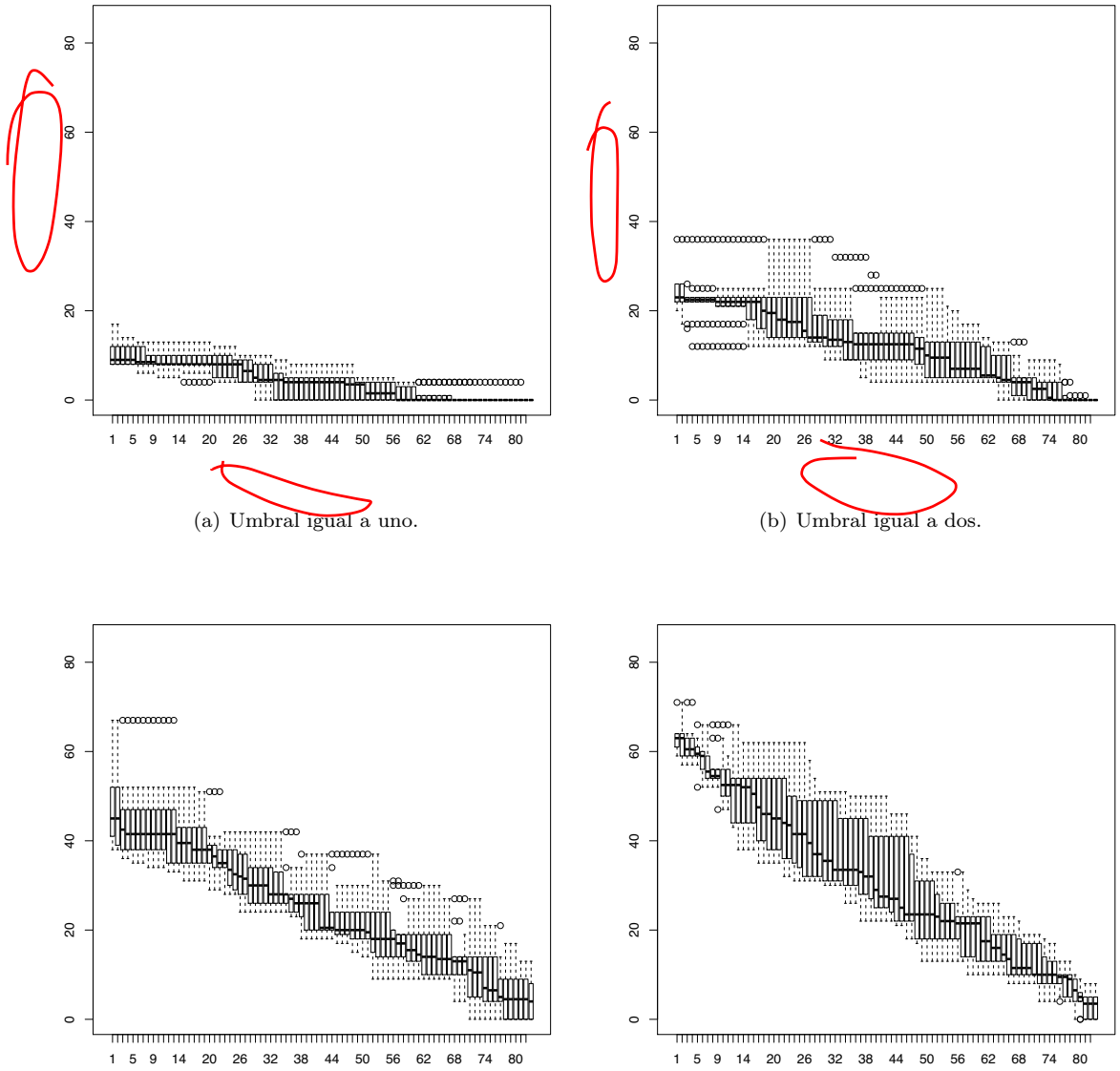


(a) Original.



(b) Con percolación.

Figura 1: Percolación de nodos para un grafo de 8×8.



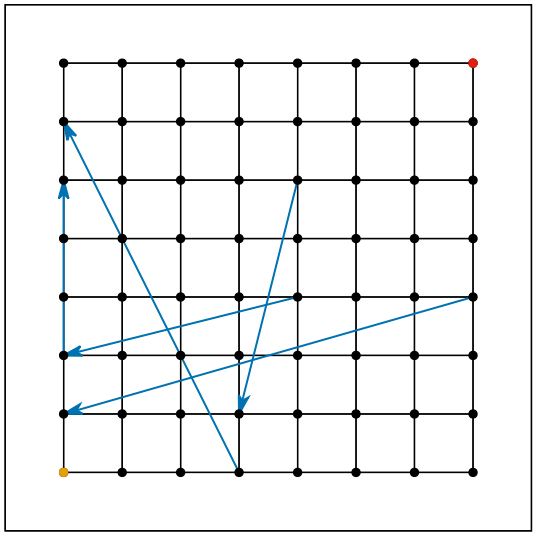
(a) Umbral igual a uno.

(b) Umbral igual a dos.

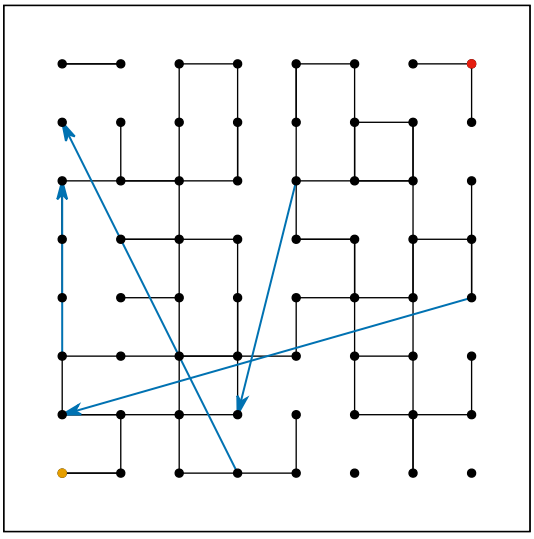
(c) Umbral igual a tres.

(d) Umbral igual a cuatro.

Figura 2: Flujo máximo percolando nodos haciendo variación en el umbral de conexiones entre nodos.

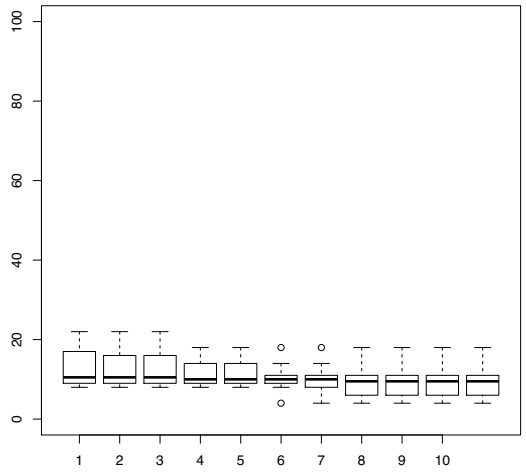


(a) Original.

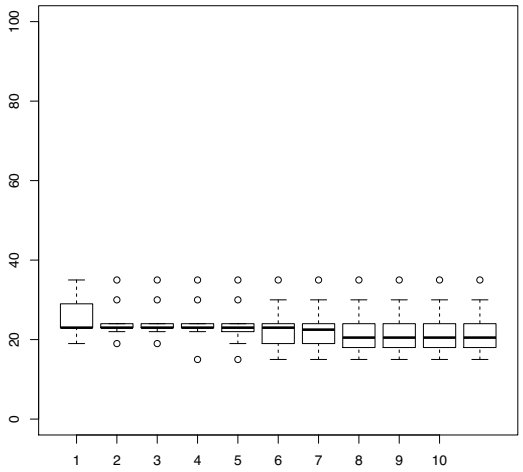


(b) Con percolación.

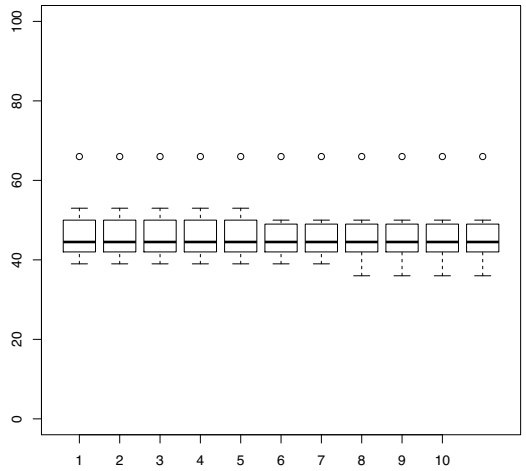
Figura 3: Percolación de aristas para un grafo de 8×8.



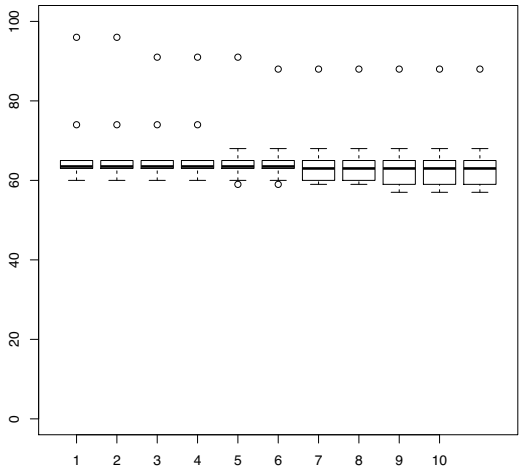
(a) Umbral igual a uno.



(b) Umbral igual a dos.



(c) Umbral igual a tres.



(d) Umbral igual a cuatro.

Figura 4: Flujo máximo percolando aristas haciendo variación en el umbral de conexiones entre nodos.

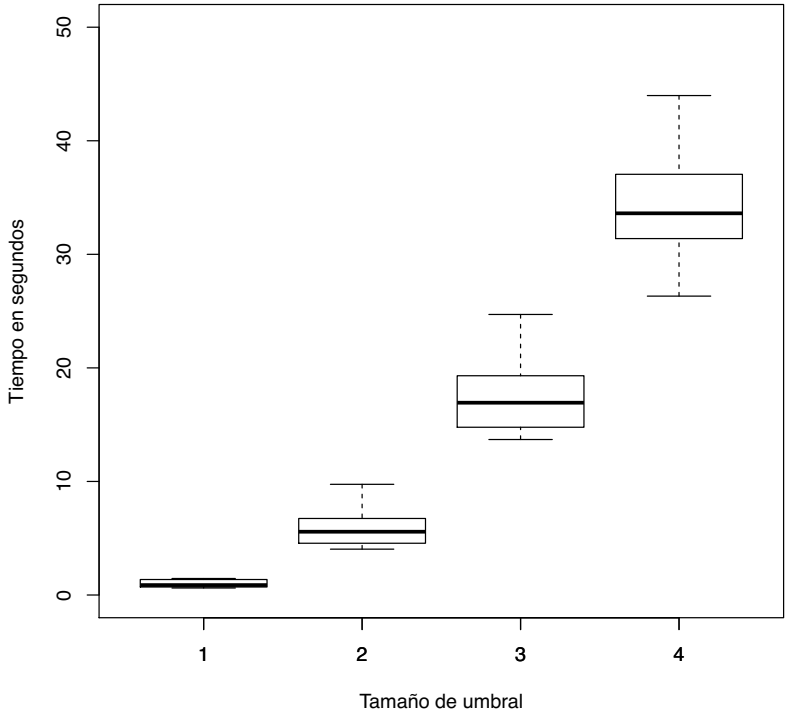


Figura 5: Tiempos de ejecución con variaciones en el umbral percolando nodos.

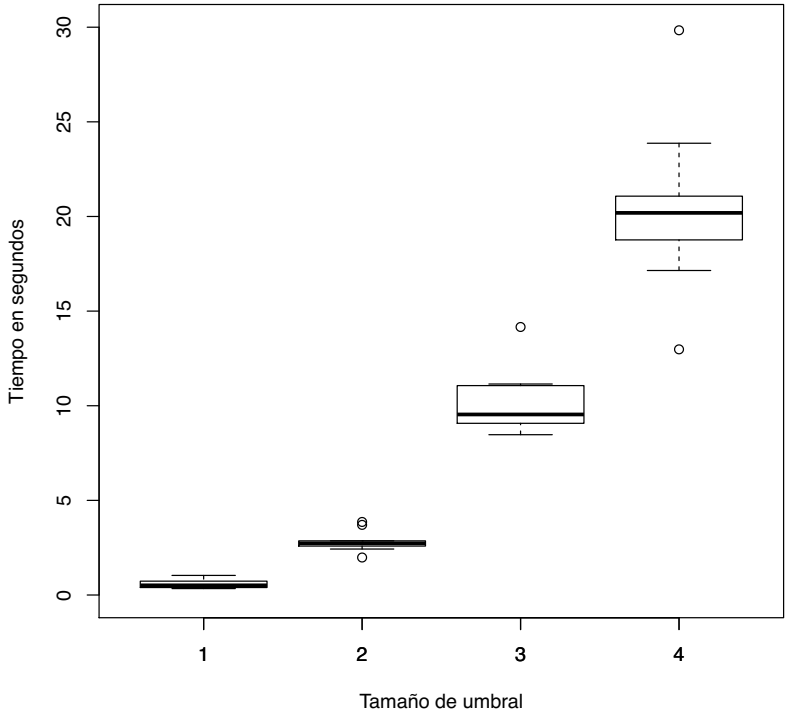


Figura 6: Tiempos de ejecución con variaciones en el umbral percolando aristas.

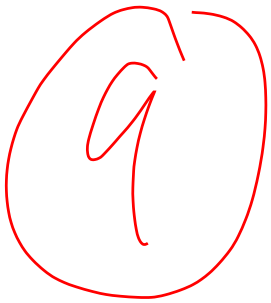
Tarea 6

Se tiene una corrección en la redacción del reporte y además se dieron sugerencias sobre el tipo de gráfica que es conveniente para analizar el flujo existente en el grafo creado. Se hizo una observación en cuanto al significado del eje que mide los tiempos de ejecución de las repeticiones generadas en la práctica y la redacción que se tuvo sobre esa gráfica.

Tarea 6
Optimización de flujo en redes

Beatriz Alejandra García Ramos

21 de mayo de 2018



1. Introducción

En esta práctica se tiene un análisis sobre el flujo máximo que se obtiene de un grafo desde su nodo inicial hasta el nodo final realizando en lugar del algoritmo de Ford-Fulkerson el algoritmo de corte mínimo. En la práctica anterior [1] únicamente se realizó un análisis en el cual se tomaba en cuenta la percolación de nodos y aristas. Ahora se tiene en cuenta la unión de nodos y sus aristas correspondientes, donde se hará un proceso hasta quedar dos nodos finales que darán como resultado el flujo máximo del grafo.

La cantidad de nodos que se tomó en cuenta para la realización el grafo es de cien nodos.

2. Flujo máximo - Corte mínimo

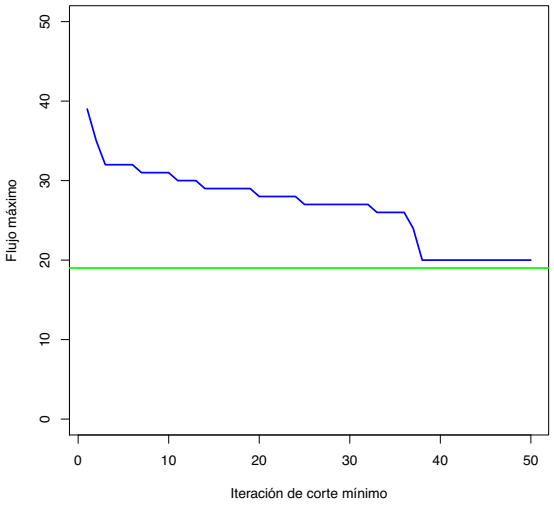
Para poder realizar el algoritmo del corte mínimo se deben simular uniones de nodos. Cuando un nodo se une con otro se olvidan esos dos nodos, se crea un nodo intermedio y la arista correspondiente entre ellos ya no se toma en cuenta; se tiene además que los vecinos correspondientes a esos dos nodos ahora son vecinos del nodo creado por la unión de ellos.

Para que se pueda lograr la unión se toma en cuenta elecciones al azar de nodos, una vez que ambos nodos se tienen se crea el nodo que está a una distancia media entre ellos dos, se elimina la arista correspondiente entre los nodos elegidos del diccionario de aristas, se unen los nodos vecinos de los dos nodos elegidos con el nuevo nodo y se crean aristas entre ellos, las aristas anteriores, aquellas que iban de los nodos elegidos a sus vecinos, se eliminan del diccionario de aristas y ahora existen nuevas aristas con el nuevo nodo. Al terminar de crear aristas se eliminan los nodos elegidos.

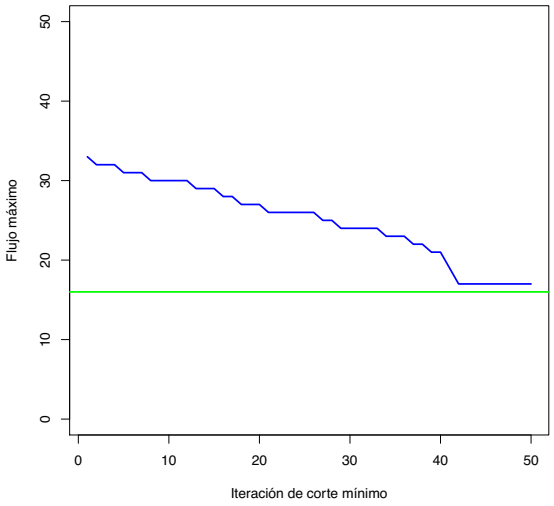
Al inicio del proceso se realiza la creación del grafo y se calcula el flujo máximo por medio del algoritmo de Ford-Fulkerson. Durante el proceso, cuando se realiza el corte mínimo, se tiene el resultado del flujo y se guarda en un archivo `.csv`, si el flujo que se va calculando es menor o igual que el anterior se guarda en el archivo, si es mayor, se vuelve a hacer el cálculo del flujo.

En la figura 1 se tiene una línea horizontal verde que representa el flujo máximo calculado con Ford-Fulkerson, la curva azúl representa los flujos que se fueron calculando al estar haciendo iteraciones para el corte mínimo. Se realizaron cinco repeticiones con cincuenta iteraciones de corte mínimo por repetición.

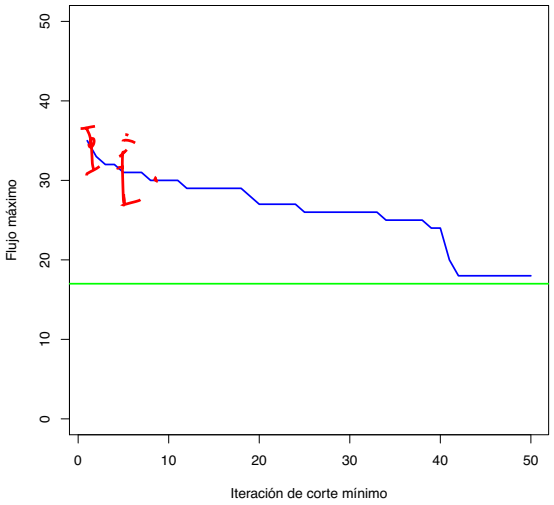
Como se puede observar, se tiene en un inicio que el flujo es muy grande con respecto al flujo máximo obtenido con el algoritmo de Ford-Fulkerson. Esto se debe a que el algoritmo de corte mínimo es una cota superior del algoritmo de Ford-Fulkerson. Existe una gran diferencia al inicio, pero conforme se van realizando las iteraciones del corte mínimo el flujo se va acercando cada vez más al valor del flujo máximo.



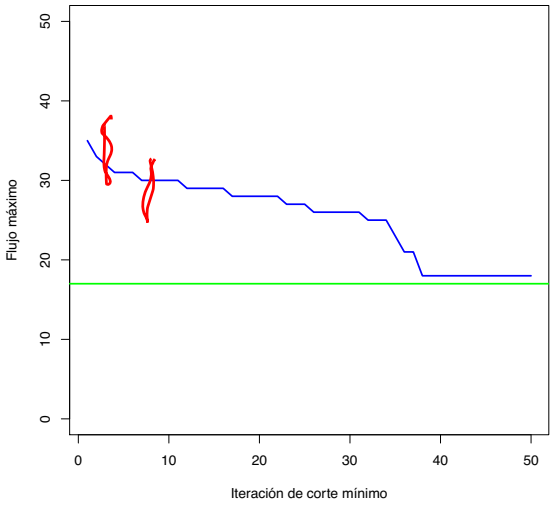
(a) Repetición 1.



(b) Repetición 2.



(c) Repetición 3.



(d) Repetición 4.

Figura 1: Flujo máximo realizando un corte mínimo.

3. Tiempos de ejecución

Los tiempos de ejecución fueron tomados en cuenta al realizar las cinco repeticiones y las cincuenta iteraciones que se obtuvieron del flujo máximo. Dado que es un proceso rápido los tiempos están medidos en segundos.

En la figura 2 se muestran las cinco repeticiones en donde las cajas de bigote tienen información sobre el tiempo que tardó en ejecutarse cada una de las cincuenta iteraciones.

Como se puede observar no existe un cambio tan notorio al realizar las repeticiones, sin embargo, existen datos atípicos en cada una de las repeticiones, que indica que existe o existen iteraciones que tardan mucho más en realizar el proceso que otras. Sin embargo, se tiene que los tiempos de ejecución del proceso de corte mínimo son pequeños, lo que es una buena señal.

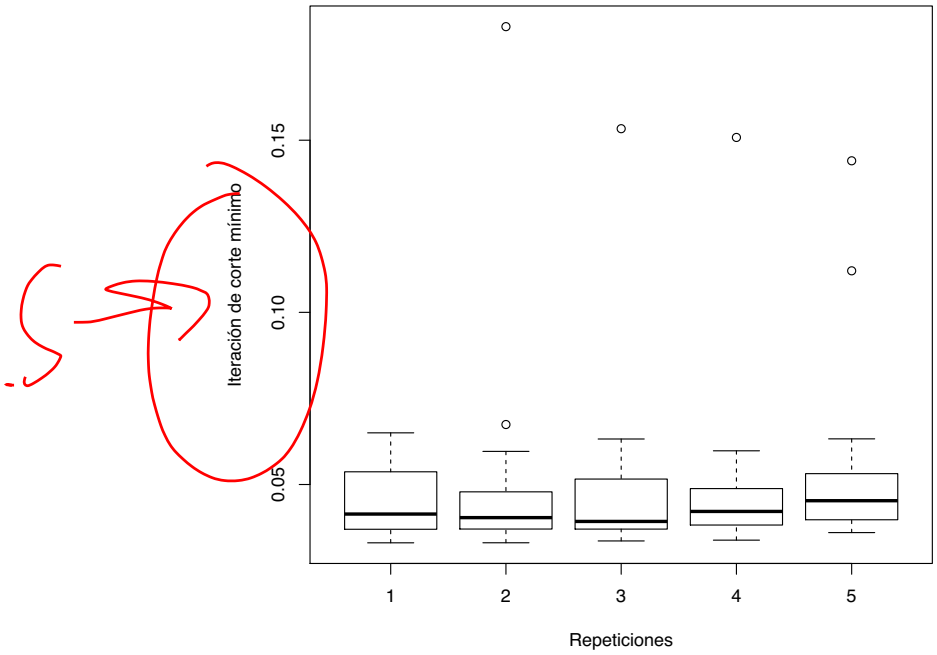


Figura 2: Tiempos de ejecución de cinco repeticiones realizando cincuenta iteraciones del corte mínimo.

Referencias

[1] Beatriz García. Tarea 5. Optimización de flujo en redes. Mayo de 2018.
https://github.com/BeatrizGarciaR/FlujoEnRedes/blob/master/Tarea5/Percolacion.py