Portafolio de Tareas
Optimización de Flujo en Redes

María Gabriela Sandoval Esquivel

Estudiante de sexto semestre del programa de
Doctorado en Sistemas Inteligentes de la
Universidad de las Américas Puebla

Mayo 2018

Tarea 1: Trazar grafos en Gnuplot

María Gabriela Sandoval Esquivel

Febrero 2018

1 Introducción

El objetivo de esta tarea es aprender a trazar grafos con nodos y aristas haciendo uso de la herramienta Gnuplot. Esta herramienta permite visualizar datos y funciones matemáticas de manera gráfica e interactiva. Las instrucciones a Gnuplot se hacen a través líneas de comando y se puede elegir entre varias terminales para el tipo de gráfico que deseamos producir.
En este reporte se incluyen primero las instrucciones para realizar grafos sencillos a partir de datos de un archivo de texto generado en Python. Después, se describen algunas de las opciones que hay para dar formato a estos grafos para representar de mejor manera los datos. Finalmente, se incluye un ejemplo de aplicación donde se visualizan las instancias y soluciones de un problema de diseño territorial.

2 Trazar Grafos

Para generar un grafo sencillo se necesita información sobre el posicionamiento de los nodos y las aristas que lo conforman. La posición de los nodos se puede describir, por ejemplo, con coordenadas en un plano; mientras que las aristas se pueden describir mediante las parejas de nodos que estas conectan.

Para crear ejemplos aleatorios de grafos el primer paso es generar los datos sobre nodos y aristas. Esto se puede hacer en un script de Python (ejemplo en el archivo `generar_datos.py`) donde se define el número de nodos cuya posición se generará de manera aleatoria y la probabilidad de unir una pareja de nodos con una arista. Los datos generados se pueden escribir en dos archivos con extensión ".dat" para que puedan ser usados por Gnuplot para generar el grafo. De estos archivos uno contiene las coordenadas de los nodos del grafo y el otro las parejas de nodos que estarán conectados.

Estos archivos deben de tener un formato muy sencillo y cada renglón representa un objeto diferente. Por ejemplo: para el archivo de nodos cada renglón representa la información de un nodo y las coordenadas de un mismo nodo se separan por espacio simple. (ejemplo en `nodos.dat`) Una vez que se tienen los

datos, se pueden indicar las instrucciones a Gnuplot para trazar el grafo. En el ambiente de Gnuplot lo primero que se debe de hacer es asegurarnos que el directorio en el que estamos trabajando sea el mismo en el que están guardados los archivos de datos. Después se establece la terminal del output que deseamos para el grafo. Hay varias opciones de terminales, por ejemplo, se puede elegir que simplemente se abra una ventana con el grafo, o que se genere una imagen formato PNG. La lista de algunas de las opciones de terminales está en: `http://www.gnuplotting.org/output-terminals/` (el resto en la documentación de Gnuplot) y se indica escribiendo en la ventana de comandos de Gnuplot:

```
set term <nombre de la terminal>
```

Después se nombra el archivo de output escribiendo con la extensión correspondiente a la terminal:

```
set output \nombrearchivo.png"
```

Para graficar los datos de un archivo usamos la función `plot` seguido de la indicación del archivo de donde se obtienen los datos y el tipo objeto con el que se graficarán los datos. Para este caso usamos el comando:

```
plot 'nodos.dat' with points pt 7
```

El tipo de símbolo que se usa para los puntos se indica con `'pt 7'`. Números diferentes dan símbolos diferentes. Para incluir más de un archivo de datos en una misma gráfica solo se necesita separar por comas a los diferentes archivos como en el siguiente ejemplo:

```
plot 'nodos.dat' with points pt 7, 'nodos1.dat' with points pt 6
```

Si se desea usar solo ciertas columnas del archive de datos se puede indicar de la siguiente manera para usar solo las columnas 1 y 3

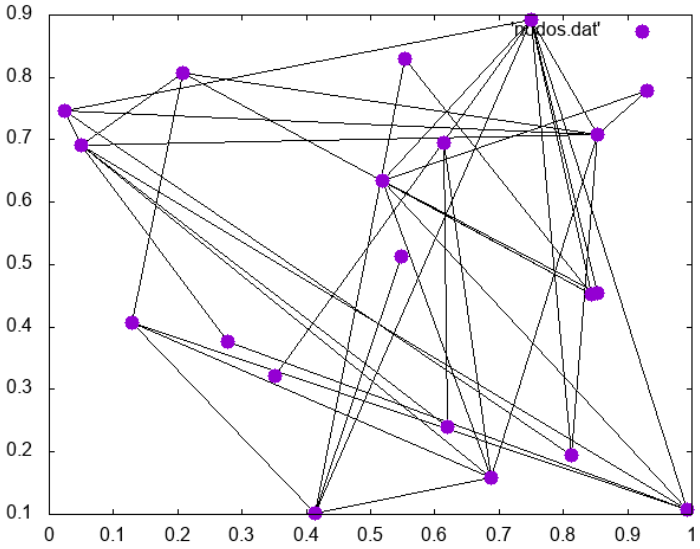
```
plot 'nodos.dat' using 1:3 with points pt 7
```

Las aristas se deben de establecer como objetos para que aparezcan en el grafo. Se establecen de la siguiente manera:

```
set arrow <arrow number> from <x1>, <y1> to <x2>, <y2> nohead
```

Donde `<arrow number>` será un índice para cada arista (se recomienda definir de manera sucesiva), se deben de indicar también las coordenadas de los nodos que une cada arista. La indicación `'nohead'` hace que las aristas no sean dirigidas. Si no se incluye, estas tendrán forma de flecha. Una vez que se definan todas las aristas se vuelve a dar la indicación de plot para que en el grafo aparezcan las aristas.

Para agilizar el proceso de generar el grafo se puede programar en Python el proceso para crear un archivo que pueda leer Gnuplot para realizar el grafo a partir de los archivos de datos. En el archivo `grafo_general.py` encontramos este programa que genera un archivo con extención `.plt`. Este último archivo se carga en Gnuplot al escribir: `load 'grafo_general.plt'`



3 Formato

Hasta ahora se tiene un grafo sencillo al que se le pueden agregar especificaciones del formato tanto como para los nodos como para las aristas.

3.1 Nodos

Para cambiar el formato simple de los nodos se usan las siguientes indicaciones:

- Tamaño: `set pointsize 2`
- Color: `plot 'nodos.dat' with points pt 7 palette cb 3`

Las indicaciones anteriores hacen que todos los nodos tengan el mismo color y el mismo tamaño. Sin embargo, podemos hacer que el tamaño y el color de los nodos describa ciertas características de los datos. Una manera de hacer que el tamaño de los nodos dependa de una tercera columna de datos, por ejemplo, es usar el objeto circles en lugar de points para graficar los nodos. La instrucción es la siguiente y la figura resultante es la 3.1

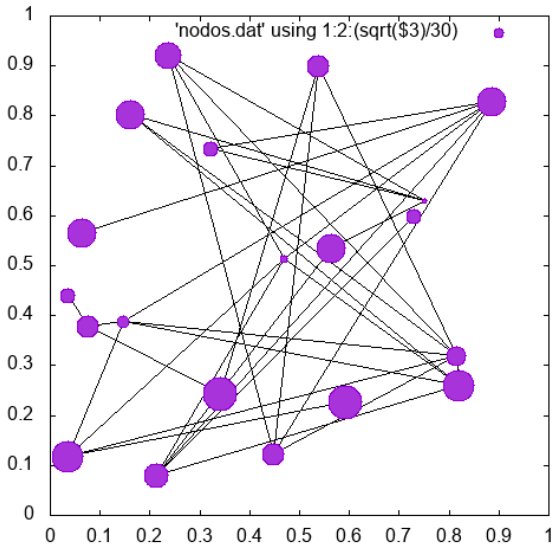
```
set style fill transparent solid 0.8
plot 'nsize.dat' using 1:2:(sqrt(\$3/50) ) with circles
```

Se pueden ajustar los valores de tamaño según convenga como lo indica la expresión `sqrt($3/50)` en la instrucción anterior.

Para cambiar el color de los nodos de acuerdo con una tercera columna de datos se define primero la paleta que se desea usar, por ejemplo:

```
set palette defined (0 'blue', 3 'green', 6 'yellow', 10 'red')
plot 'nodosr.dat' using 1:2:3 with points pt 7 palette
```

Figure 1: Ejemplo de grafo con nodos de diferente tamaño



Para combinar tamaños y colores que dependan de una tercera y cuarta columna respectivamente se puede hacer con la siguiente instrucción:

```
plot ‘nodosr.dat’ using 1:2:(sqrt(\$3/50) ):4 with circles palette
```

3.2 Aristas

El formato que se le puede dar a las aristas es definir indentación, con la siguiente instrucción:

```
set arrow 1 dashtype 2
```

También se puede cambiar el grosor de las aristas de la siguiente manera:

```
set style arrow 1 linewidth 3
```

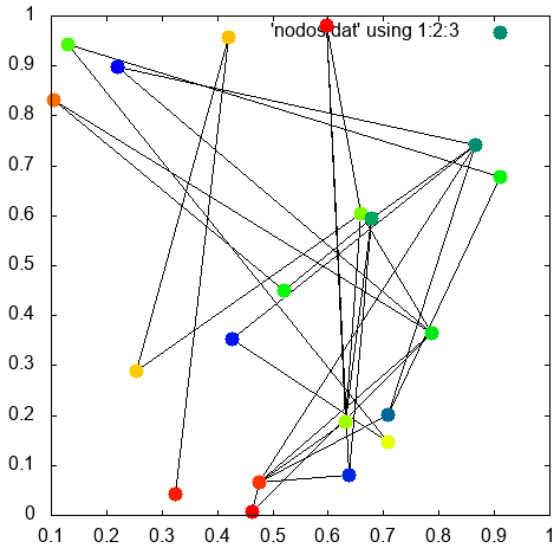
Para cambiar el color:

```
Set arrow 1 linecolor ‘blue’
```

3.3 Ventana

- Quitar título: `plot 'nodos.dat' with points pt 7 notitle`
- Quitar la caja de color: `unset color box`
- Definir el rango de los ejes:
  ```
  set xrange [0,200]
  set yrange [0,200]
  ```

Figure 2: Ejemplo de grafo coloreado



4 Aplicación

Al aprender sobre esta herramienta me di cuenta de que me puede ser muy útil para visualizar instancias y soluciones del problema en el que estoy trabajando para mi tesis. Se trata de un problema de diseño territorial en el que se busca dividir un área geográfica en territorios de acuerdo con ciertos criterios de planeación.
La formulación matemática para este problema considera al área geográfica como un conjunto de nodos que están conectados. Los nodos representan unidades básicas que al agruparse forman territorios, pueden ser por ejemplo colonias, municipios o manzanas. Las unidades básicas tienen un cierto tamaño de acuerdo con medidas de actividad como número de habitantes o capacidad económica. Los nodos se conectan por aristas si las unidades básicas son vecinos geográficos debido a que se busca que los territorios que se formen sean contiguos y compactos.
Hice un programa en Python que usa los datos que tengo para mis instancias para crear un archivo con las instrucciones necesarias para que Gnuplot grafique las unidades básicas en el plano con círculos que representan el tamaño del distrito y los nodos indican vecindad. (Los archivos correspondientes son: instancias.py e instancias.plt)
Las figuras 3 y 4 muestran ejemplos de las gráficas de las instancias que uso para hacer las pruebas de mi algoritmo.

Las soluciones que se obtienen indican a qué territorio pertenece cada unidad básica. Pude usar esta información para que el color de cada nodo represente el territorio al que pertenece.El programa de Python que usa información de

Figure 3: Instance 2DU60-05-2

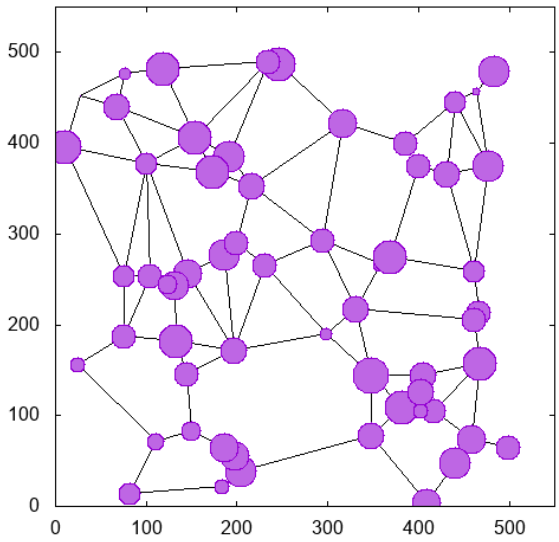


Figure 4: Instance 2DU60-05-3

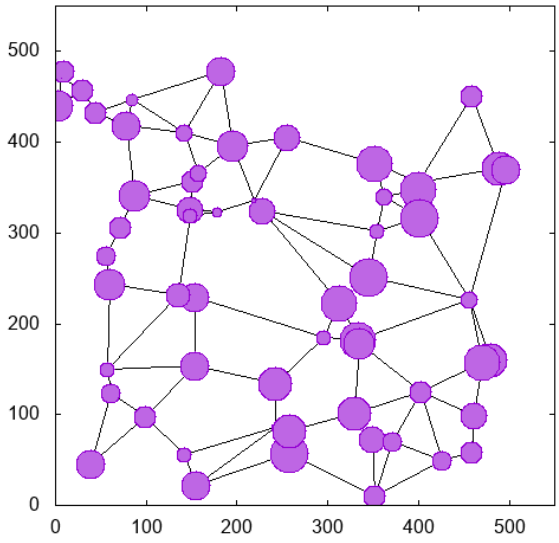
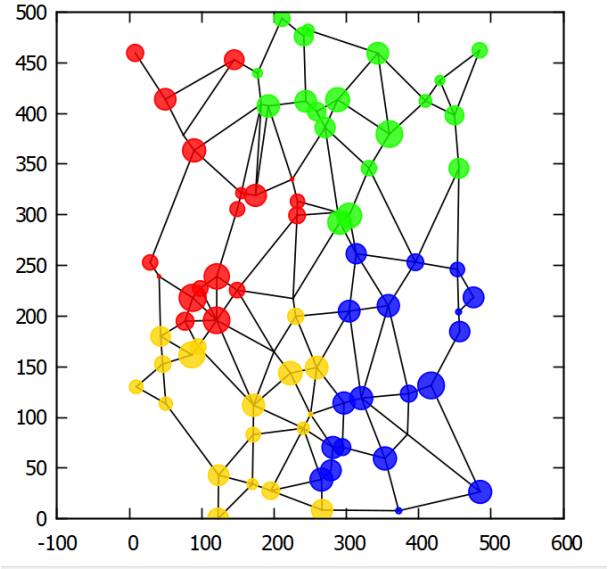


Figure 5: Solution 2DU80-05-1



las instancias y las soluciones para generar un grafo con nodos de tamaños diferentes (según la actividad) y colores diferentes (según al territorio al cual son asignados) se muestra en el apéndice de este reporte. En la figura 5 se muestra un ejemplo de la gráfica de una solución.

5 Conclusión

Con esta actividad pude aprender a usar varias funciones de Gnuplot para trazar grafos con características especiales. También descubrí que es útil generar los archivos con instrucciones para Gnuplot mediante un programa en Python. Esto me resulta especialmente útil para analizar los reultados de los algoritmos que estoy desarrollando para mi tema de tesis. En especial será útil para visualizar las soluciones parciales que resultan en cada fase de los algoritmos iterativos. De esta manera será fácil analizar cómo evolucionan las soluciones y el tipo de cortes que se están realizando. En un principio me resultó dificil adaptarme al nuevo ambiente de trabajo con estas herramientas, pero su utilidad hacen que valga la pena.

6 Apéndice: Código para trazar soluciones

```
n = 80
p = 4
center = []

basicUnit = []
neighbors = []
territory = []
terr = [None]*n
BU_file = "psol80-05-2.dat"

with open("2DU80-05-2.dat",'r') as archivo:
        n = int(archivo.readline())
        for i in range(n):
            stringLine = archivo.readline()
            splitLine = stringLine.split(" ")
            valueList = [float(e) for e in splitLine]
            index, x, y, a, b, c = valueList
            basicUnit.append((x,y,a))
        neigh = int(archivo.readline())
        for i in range(neigh):
            stringLine = archivo.readline()
            splitLine = stringLine.split(" ")
            valueList = [int(e) for e in splitLine]
            neigh1, neigh2 = valueList
            neighbors.append((neigh1,neigh2))

archivo.close()

with open("sol80-05-2.txt",'r') as solucion:
    for i in range(p):
        c = int(solucion.readline())
        center.append(c)
    for i in range(p):
        string_terr = solucion.readline()
        splitLine = string_terr.split(" ")
        splitLine = splitLine[0:(len(splitLine)-1)]
        valueList = [int(e) for e in splitLine]
        territory.append(valueList)
        for j in range(len(splitLine)):
            idx = valueList[j]
            #print(idx)
            terr[idx-1]= i+1
solucion.close()
```

```
with open("psol80-05-2.dat",'w') as psolucion:
    for i in range(n):
        x,y,size= basicUnit[i]
        print(x,y,size,terr[i], file = psolucion)
psolucion.close()

with open("solution.plt",'a') as aristas:
        print("set term png", file = aristas)
        print("set output 'psol.png' ", file = aristas)
        print("set pointsize 1", file = aristas)
        print("set size square", file = aristas)
        print("set style fill transparent solid 0.8", file = aristas)
        print("set palette defined (0 'blue', 3 'green', 6 'yellow', 10 'red') "
        print("unset colorbox", file = aristas)
        num = 1
        for i in range(neigh):
            neigh1, neigh2 = neighbors[i]
            x1, y1, s1 = basicUnit[neigh1]
            x2, y2, s2 = basicUnit[neigh2]
            print("set arrow", num, "from", x1, "," ,y1,"to", x2, ",", y2, "no
            num +=1

        print("plot 'psol80-05-2.dat' using 1:2:(2*sqrt(\$3-600)):4 with circles
        print("unset arrow", file = aristas)
        print("unset output", file = aristas)
aristas.close()
```

Tarea 2: Grafos simples, ponderados y dirigidos

María Gabriela Sandoval Esquivel

Febrero 2018

1 Introducción

El objetivo de esta tarea es mejorar el programa que tenemos para trazar grafos en Gnuplot [1] incorporando el uso de clases para agilizar el proceso y añadiendo características de estilo a las aristas. Las clases se definen en nuestro programa de Python [2] para poder crear objetos con los atributos deseados para los grafos. Su uso agiliza el proceso para generar archivos con instrucciones para graficar diferentes tipos de grafos en Gnuplot.

2 Estilos aristas

Los grafos simples, ponderados y dirigidos se distinguen por los atributos que se les dan a sus aristas. En un grafo simple las aristas representan conexiones reflexivas entre dos nodos y no se considera ninguna otra característica que distinga a una arista de otra. Visualmente las aristas son líneas simples que conectan a dos nodos y todas las aristas de un grafo simple son iguales. La instrucción para trazar las aristas de un grafo simple en Gnuplot es la siguiente:

```
set arrow <arrow_number> from <x1,y1> to <x2,y2> nohead
```

En un grafo ponderado a las aristas se les asignan diferentes pesos y visualmente esto se puede representar cambiando el grosor de la arista según su peso. La instrucción en Gnuplot sería la siguiente:

```
set arrow <arrow_number> from <x1,y1> to <x2,y2> nohead lw <arrow_width>
```

Finalmente, un grafo dirigido indica que hay diferencia entre los nodos que conectan un grafo. Por ejemplo, un nodo puede ser el nodo de salida mientras que el segundo el de llegada. De esta manera se distingue entre la arista que va del nodo 'a' al nodo 'b' y la que va del nodo 'b' al 'a'. Visualmente las aristas de grafos dirigidos se representan con flechas y para trazarlas en Gnuplot se usa la siguiente instrucción:

```
set arrow <arrow_number from <x1,y1> to <x2,y2> head
```

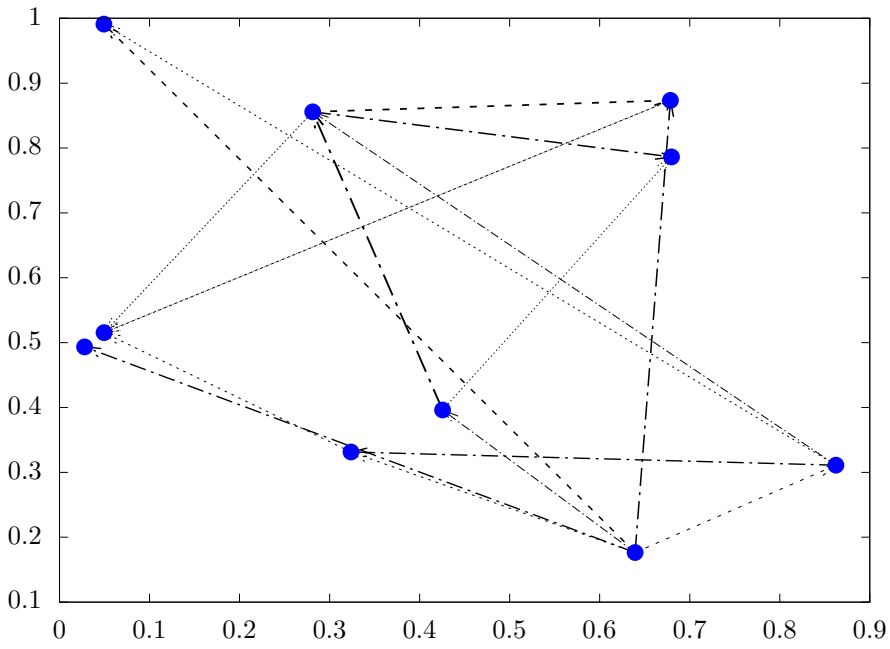


Figure 1: Ejemplos de estilos de aristas

Se puede indicar también si deseamos que la fleche tenga diferentes estilos con las siguientes instrucciones:

```
set arrow <arrow_number from <x1,y1> to <x2,y2> head filled\\
set arrow <arrow_number from <x1,y1> to <x2,y2> head nofilled\\
set arrow <arrow_number from <x1,y1> to <x2,y2> head empty
```

3 Definir Clases

En programación orientada a objetos se hace uso de clases para definir tipos de objetos. Al crear una clase se pueden definir los campos (o variables) y los métodos que estarán relacionados a los objetos que pertenezcan a esa clase [3] Los campos (fields en inglés) son atributos de los objetos de esa clase. Al crear la clase de Grafos, por ejemplo, se puede definir el campo de los nodos y el campo de las aristas. Los métodos son funciones que se aplican a los objetos de esa clase. En este caso tendremos funciones para agregar nodos y aristas al grafo, entre otras.

3.1 Campos

Como ya vimos los campos básicos para la clase de grafos son los nodos y las aristas. En el programa definimos a los nodos como un conjunto y a las aristas como diccionarios que relacionan a los elementos del conjunto de nodos. Los nodos los describimos con un nombre (variable tipo string) y se les atribuye un par de coordenadas, y variables numéricas para asignar su tamaño y su color en caso de que sea necesario en el grafo. Las aristas por su parte se describen como una pareja de nodos, se les atribuye una variable de peso (variable numérica), se indica si son dirigidas (variable booleana) y el estilo de la línea punteada (variable entera). Otro campo que se define para los grafos es el de vecinos que es un diccionario que relaciona a cada nodo con el conjunto formado por los nodos que están conectados con él por una arista.

3.2 Métodos

Los métodos de una clase son funciones que se definen para sus objetos y atributos. A continuación, está la descripción de las funciones que se han definido hasta ahora para la clase de grafos.

3.2.1 Agrega

Esta función agrega elementos al conjunto de nodos indicando las características de posición, tamaño y color. Si no se desea especificar alguna de estas características se asignan valores por default para un nodo simple.

```
def agrega(self, v,tamano = 1, posicion = (0,0), color = 0):
    self.nodos.add(v)
    self.tamano[v] = tamano
    self.posicion[v] = posicion
    self.color[v] = color
    if not v in self.vecinos:
        self.vecinos[v] = set()
```

3.2.2 Conecta

Con esta función se indica que existe una arista entre dos nodos y las características específicas de esa arista. Se puede hacer uso de esta función con nodos que no hayan sido definidos previamente, en este caso la función conecta llama a la función agrega con valores default para las características de los nodos.

```
def conecta(self, v, u, peso=1, dirigido = False, tipo = 1):
    if not v in self.nodos:
        self.agrega(v)
    if not u in self.nodos:
        self.agrega(u)
    self.vecinos[v].add(u)
    self.vecinos[u].add(v)
    if dirigido:
        self.aristas[(u, v)] = (peso, tipo, dirigido) # en ambos
    sentidos
```

```
    else:
        self.aristas[(v, u)] = self.aristas[(u, v)] = (peso, tipo,
    dirigido)
```

3.2.3 Complemento

Esta función se usa para definir otro objeto de la clase Grafo. El grafo complemento contiene el mismo conjunto de nodos que el original, pero sus aristas corresponden a las que no existen en el grafo original.

```
    def complemento(self):
        comp= Grafo()
        for v in self.nodos:
            for w in self.nodos:
                if v != w and (v, w) not in self.aristas:
                    comp.conecta(v, w, 1)
        return comp
```

3.2.4 Aleatorio

Con esta función se crea un grafo especificando un número de nodos deseado y una probabilidad de conexión. De este modo se creará una arista entre todo par de nodos de acuerdo a esta probabilidad.

```
def aleatorio(self, n=10, prob = 0.5, dirigido = False,
    rand_style_aristas = False):
    from random import random
    for i in range(n):
        tag_nodo = str(i)
        x = random()
        y = random()
        size = random()
        color = random()
        self.agrega(tag_nodo, size, (x,y),color)
    for i in range(n - 1):
        for j in range(i + 1, n):
            if random() < prob:
                if rand_style_aristas:
                    atipo = int(10*random()) % 5
                    peso = 3*random()
                else:
                    atipo = 1
                    peso = 1
                self.conecta(str(i),str(j),peso, dirigido,atipo)
```

3.2.5 Lee

Con esta función se crea un grafo de acuerdo con datos de un archivo de texto con un formato específico. La primera línea de ese archivo debe de indicar el número de nodos y los renglones siguientes deben de indicar las características de esos nodos. La primera columna corresponde al índice del nodo, las siguientes dos las coordenadas de posición del nodo seguidas de la columna que indica el

tamaño y finalmente el color. Después de la información de los nodos sigue un renglón que indica el número de aristas y después los renglones que indican las parejas de índices de los nodos que forman a cada arista. Este formato es el que tienen las instancias que uso para mi proyecto de tesis y con el que puedo visualizar este tipo de datos.

```
def leer(self, filename = "2DU80-05-2.dat"):
    with open(filename, 'r') as archivo:
        nn = int(archivo.readline())
        for i in range(nn):
            stringLine = archivo.readline()
            splitLine = stringLine.split(" ")
            valueList = [float(e) for e in splitLine]
            index, x, y, a, b, c = valueList
            index = int(index)
            idx = str(index)
            #print(idx)
            a = pow((a - 590),3)
            self.agrega(idx, a, (x,y), 1)
        neigh = int(archivo.readline())
        for i in range(neigh):
            stringLine = archivo.readline()
            splitLine = stringLine.split(" ")
            neigh1, neigh2 = splitLine
            neigh2, basura = neigh2.split("\n")
            self.conecta(neigh1,neigh2,1)
    archivo.close()
```

3.2.6 Gnuplot

Finalmente está la función que se nombró gnuplot que sirve para crear un archivo con las instrucciones necesarias para que se trace el grafo con las características deseadas. A esta función se le puede indicar el nombre del archivo y si se desea que se consideren los colores y tamaños de los nodos para la visualización. En esencia esta función contiene la mayoría de las instrucciones de la versión anterior del programa para trazar grafos.

```
def gnuplot(self, gcolor = True, gtamano = True, name = "grafo_a"):
    n = len(self.nodos)
    with open("nodos.dat",'w') as archivo_nodos:
        for nodo in self.nodos:
            #nodo = str(i)
            (x,y) = self.posicion[nodo]
            size = self.tamano[nodo]
            color = self.color[nodo]
            print(x, y,size,color, file = archivo_nodos)
    filename = name + ".plt"
    imagename = " '"+ name + ".tex' "
    arrow_idx = 1
    archivo_nodos.close()
    with open(filename, 'w') as archivo:
        print("set term epslatex", file = archivo)
        print("set output"+imagename, file = archivo)
        print("set pointsize 2", file = archivo)
        print("unset arrow", file = archivo)
```

```
        print("unset colorbox", file = archivo)
        for (ii,jj) in self.aristas:
            (x1, y1) = self.posicion[ii]
            (x2, y2) = self.posicion[jj]
            (apeso, atipo, adirected) = self.aristas[(ii,jj)]
            head = "nohead"
            if adirected:
                head = "head"
            print("set arrow", arrow_idx, "from", x1, "," ,y1," to
", x2, ",", y2, head, " lw ",apeso, " dashtype ", atipo, file =
 archivo)
            arrow_idx += 1
        #print("set style fill transparent solid 0.7", file =
archivo)
        print("set style fill solid", file = archivo)
        print("set palette defined (0 'blue', 3 'green', 6 'yellow
', 10 'red') ", file = archivo)
        if gcolor and gtamano:
            print("color y tama o!!")
            print("plot 'nodos.dat' using 1:2:(sqrt(3)/30):4 with
circles palette notitle", file = archivo)
        elif gcolor:
            print("sin tama o!!")
            print("plot 'nodos.dat' using 1:2:3 with points pt 7
palette notitle", file = archivo)
        elif gtamano:
            print("sin color!!")
            print("plot 'nodos.dat' using 1:2:(sqrt(3)/30) with
circles notitle", file = archivo)
        else:
            print("sin color sin tama o!!")
            print("plot 'nodos.dat' using 1:2 with points pt 7 lc
rgb 'blue' notitle", file = archivo)

        archivo.close()
```

4 Conclusión

Definir una clase para definir grafos como objetos tiene varias ventajas tanto para la manipulación de estos datos en Python como para crear los archivos de instrucción para gnuplot de manera más eficiente. La estructura que se definió aquí para los objetos de la clase grafo es útil para manejar distintos atributos y sus relaciones dentro de Python. Por ejemplo, se puede acceder fácilmente al conjunto de vecinos de un nodo en específico o al conjunto de aristas de un grafo solamente. Esto resultó útil para indicar la instrucción del trazo de aristas y puede ser útil en varias otras aplicaciones. Por otro lado, esta estructura y sus funciones permiten que fácilmente se puedan definir o leer varios grafos con sus características específicas y crear las instrucciones para trazar grafos con diferentes formatos en pocas líneas de código.

References

[1]

[2] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[3] Swaroop C. H. *A Byte of Python*. CreateSpace Independent Publishing Platform, USA, 2015.

Tarea 3: Medición experimental de la complejidad asintótica

María Gabriela Sandoval Esquivel

Abril 2018

1 Introducción

El objetivo de esta tarea es analizar el desempeño de dos algoritmos que tienen diferente complejidad asintótica: el algoritmo Floyd-Warshall y el algoritmo Ford-Fulkerson. El primero encuentra el camino más corto entre todos los pares de nodos en un grafo y fue propuesto en 1962 por Robert Floyd [1] y Stephen Warshall [2]. El segundo encuentra el flujo máximo entre un par de nodos haciendo uso de caminos aumentantes y fue propuesto por Lester Ford y Dester Fulkerson en 1965.

La teoría de la complejidad computacional pretende medir la efectividad de los algoritmos. Existen varias herramientas de análisis para describir la efectividad de los algoritmos y éstas se basan en las operaciones elementales que se realizan en el algoritmo. Una notación comúnmente utilizada en la literatura para describir la complejidad computacional de un algoritmo es la notación de la O grande (en inglés: Big O Notation)[3]. Se dice que un algoritmo corre en tiempo $O(f(n))$ si para ciertas constantes c y n_0, el tiempo que le toma al algoritmo es a lo más $cf(n)$ para todo $n >= n_0$[3]. En esta definición se considera que n es un parámetro que indica el tamaño del problema (por ejemplo: el número de nodos del grafo).

Para interés de esta práctica, se sabe que el algoritmo de Floyd-Warshall se ejecuta en tiempo $O(n^3)$[3] y que el algoritmo de Ford-Fulkerson se ejecuta en tiempo $O(mF)$ donde m es el número de aristas del grafo y F es el flujo máximo del grafo.

En esta práctica se hace una medición experimental de la complejidad de estos algoritmos. Primero se añadieron las funciones de estos algoritmos a la clase de Grafo que se realizó para la práctica pasada. Con estas funciones se hicieron pruebas para dos tipos de grafos: dirigidos y no dirigidos, ambos con pesos en sus aristas que indican el flujo. En este reporte se incluyen las comparaciones de los tiempos de ejecución de ambas funciones para grafos de diferente tamaño. Se pretende verificar que estos resultados coincidan con la complejidad computacional de los algoritmos.

2 Programación de algoritmos

Estas funciones están basadas en las que se encuentran en la página del curso de Matemáticas Discretas [4] y adaptadas a la estructura de grafos de esta clase. La función del algoritmo de Floyd-Warshall da como resultado un diccionario que relaciona a cada par de nodos del grafo con el camino más corto entre ellos. La función del algoritmo de Ford-Fulkerson tiene como resultado el flujo máximo entre el par de nodos que recibe la función. Además, se añadieron contadores de tiempo para medir el tiempo de ejecución de cada función. De este modo, cada función tiene como segundo resultado su tiempo de ejecución.

3 Pruebas experimentales

En esta sección se presentan los resultados para los diferentes tipos de grafos: dirigidos y no dirigidos. Para cada tipo se generan grafos de diferentes tamaños de manera aleatoria: cada par de nodos se conecta con una probabilidad del 30% y con un peso aleatorio. Para cada grafo se generaron 500 instancias para hacer las pruevas de los algoritmos. A continuación se presentan gráficas de caja y bigotes para cada algoritmo y tipo de grafo. En el eje x se muestran el número de nodos que tenían las instancias y en el eje y el tiempo de ejecución en segundos.

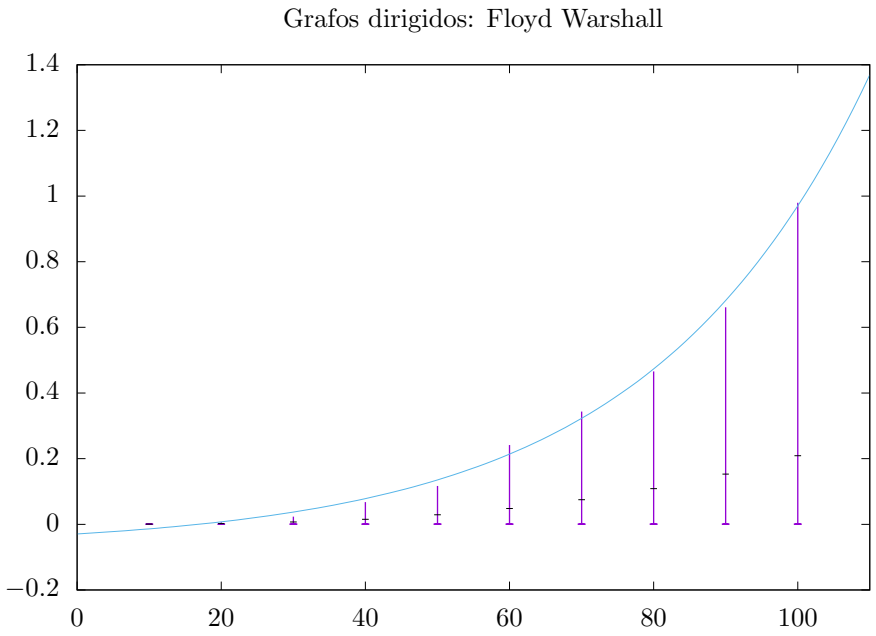


Figure 1: Diagramas de cajas y bigotes para el algoritmo de Floyd-Warshall y función exponencial ajustada a los tiempos máximos de ejecución

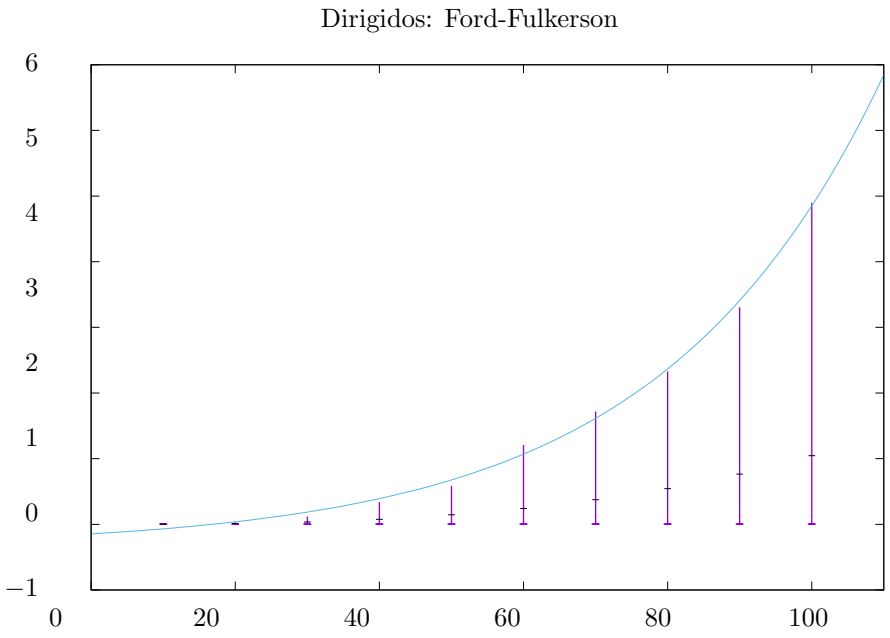


Figure 2: Diagramas de cajas y bigotes para el algoritmo de Ford-Fulkerson y función exponencial ajustada a los tiempos máximos de ejecución

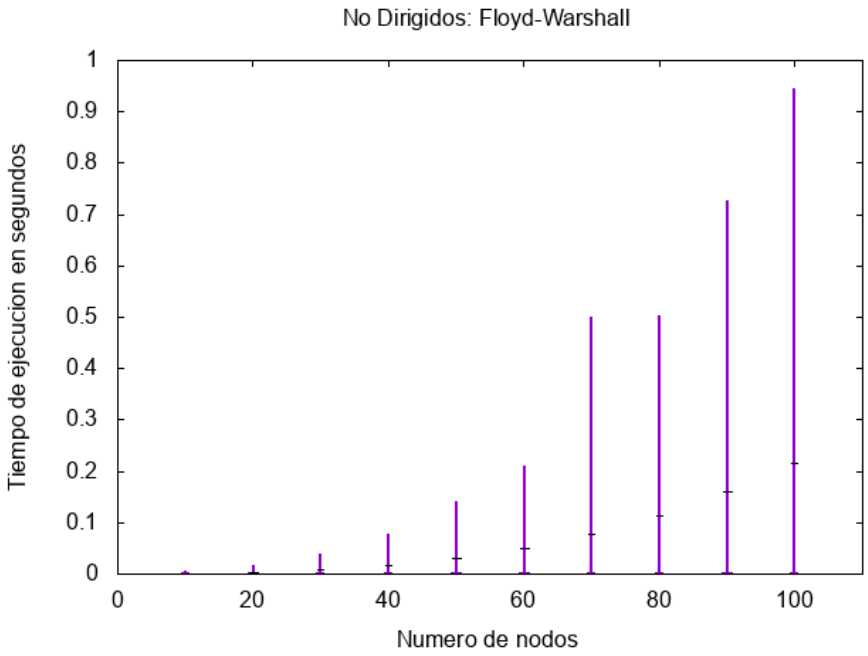


Figure 3: Diagramas de cajas y bigotes para el algoritmo de Floyd-Warshall y función exponencial ajustada a los tiempos máximos de ejecución

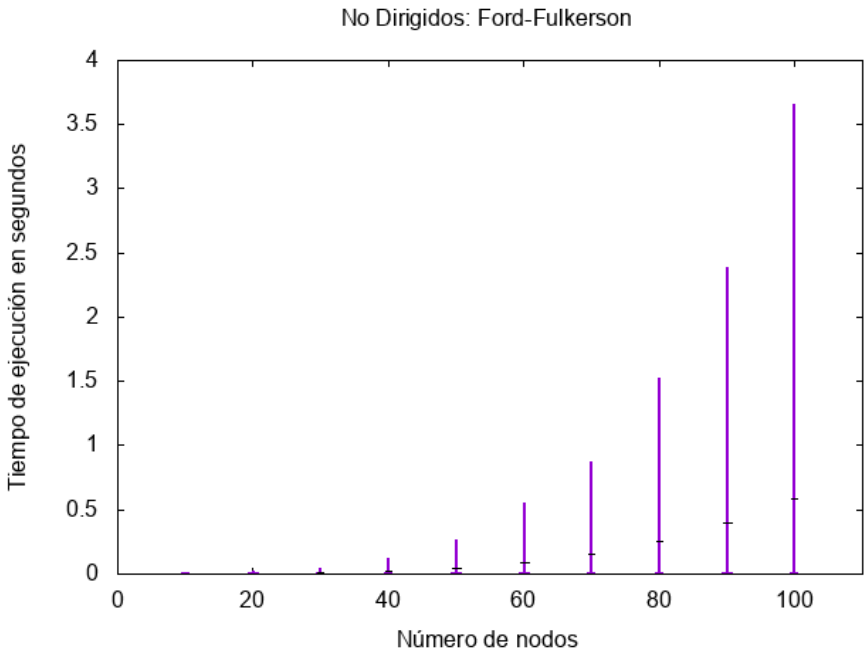


Figure 4: Diagramas de cajas y bigotes para el algoritmo de Ford-Fulkerson y función exponencial ajustada a los tiempos máximos de ejecución

4 Conclusiones

Es evidente en las cuatro gráficas anteriores que los resultados para los dos algoritmos y cada tipo de gráfico crece de manera exponencial de acuerdo al número de nodos (y por consecuencia de aristas) que tienen las instancias generadas. En cuanto a la diferencia entre los algoritmos, se puede ver que para ambos tipos de grafos los tiempos de ejecución crecen más rápido para el algoritmo de Ford-Fulkerson. Esta observaión concuerda con la teoría. Por otro lado, en cuanto a los tipos de grafos los tiempos de ejecución son menores para los grafos no dirigidos en ambos algoritmos. La diferencia es más notoria en el algoritmo de Ford-Fulkerson y puede justificarse por el hecho de que hay un menor número de aristas en los grafos no dirigidos por la manera en la que fueron generados.

References

[1] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[2] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[3] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network flows*. Elsevier, 2014.

[4] Satu Elisa Schaeffer. Matemáticas discretascurso en línea.

Tarea 4: Experimentos con el algoritmo de Floyd-Warshall

María Gabriela Sandoval Esquivel

Abril 2018

1 Introducción

El algoritmo de Floyd-Warshall que se desarrolló en la tarea anterior da como resultado una matriz con las distancias entre todo par de nodos dentro de un grafo [1]. Este resultado es útil para calcular una propiedad estructural de los grafos: la distancia promedio d. La distancia promedio se obtiene sumando todos los elementos de la matriz de Floyd-Warshall y dividiendo el resultado por el número de nodos al cuadrado.

Otra propiedad estructural de los grafos es el coeficiente de agrupamiento. Éste se define para cada nodo del grafo midiendo la densidad del grafo inducido por su vecindad. En este sentido, el coeficiente de agrupamiento define qué tan crucial es cada nodo para mantener conectados a los nodos en su vecindad. En contraste con la distancia promedio, que es una medida global del grafo, el coeficiente de agrupamiento define una propiedad local.

Estas dos propiedades varían conforme incrementa la probabilidad de que un par de nodos se conecte dentro de un grafo. En este reporte se incluyen algunos ejemplos que ilustran esta variación para diferentes tipos de grafos. Estas medidas son útiles en la práctica para analizar características de mundo pequeño (*small-world*) en redes como por ejemplo para evaluar índices de infección de algún virus [2].

En esta tarea se programaron funciones para construir grafos circulares con ciertas propiedades de conectividad y para calcular las medidas de distancia promedio y coeficiente de agrupamiento. Se hicieron pruebas para evaluar los resultados de estas dos medidas conforme incrementaba la probabilidad de conectar cada par de nodos en el grafo. Así mismo se hicieron más pruebas del desempeño del algoritmo de Floyd-Warshall, en cuanto a su complejidad computacional, con los grafos circulares donde cada nodo tiene al menos dos vecinos.

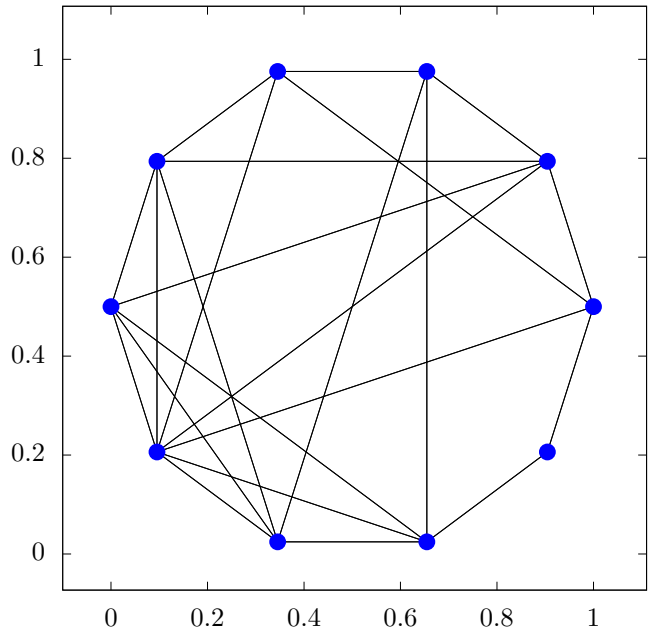


Figura 1: Ejemplo de grafo circular con 10 nodos.

2 Programación de funciones

Primero se creó una función para generar grafos circulares donde se especifica el mínimo número de vecinos que tiene cada nodo k y la probabilidad de generar vecinos adicionales p. Para esta función los nodos se posicionen equitativamente espaciados en un círculo unitario. Después se conecta cada nodo con sus k vecinos más cercanos y para el resto de las parejas de nodos se decide si se conectan de acuerdo con la probabilidad de p. En la figura 1 se ilustra un grafo de este tipo con 10 nodos, un k igual a uno y una probabilidad p igual a 2^{-2}.

Para la función que calcula la distancia promedio simplemente se suman todos los elementos de la matriz de Floyd-Warshall y se divide el resultado por el número de nodos al cuadrado. Por otro lado, para la función del coeficiente de agrupamiento se calcula para cada nodo la densidad del grafo inducido por su vecindad. La densidad del sub-grafo se calcula dividiendo el número de aristas del grafo entre el máximo número de aristas posibles.

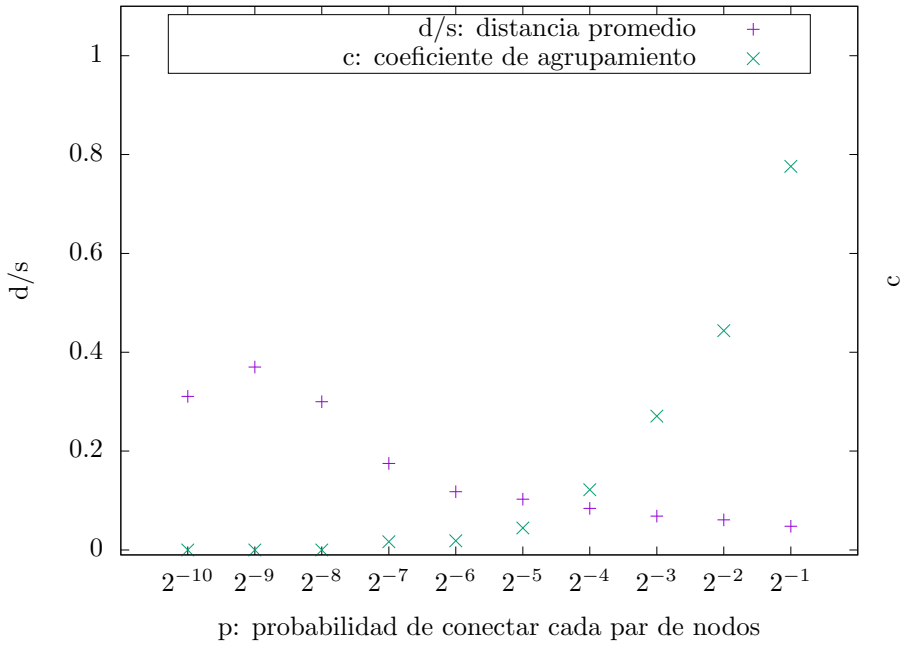


Figura 2: Resultados para k = 1

3 Pruebas experimentales

A continuación, se presentan gráficas que muestran el comportamiento de las propiedades estructurales conforme crece la probabilidad de conectar cada par de nodos en escala logarítmica. Estas pruebas se hacen en grafos circulares de 50 nodos generados aleatoriamente variando p y k. En estas gráficas el coeficiente de agrupamiento ya tiene un rango entre cero y uno, pero para normalizar la distancia promedio se calcula la distancia máxima posible en un grafo de esas características. La distancia máxima se define por la siguiente fórmula:

$$(n/2)/k$$

Esta fórmula se deriva se deriva del razonamiento de que, si en un grafo los nodos se conectan con sus k nodos más cercanos, los nodos más alejados se encuentran opuestos en el círculo y así se obtiene el camino más corto entre ellos.

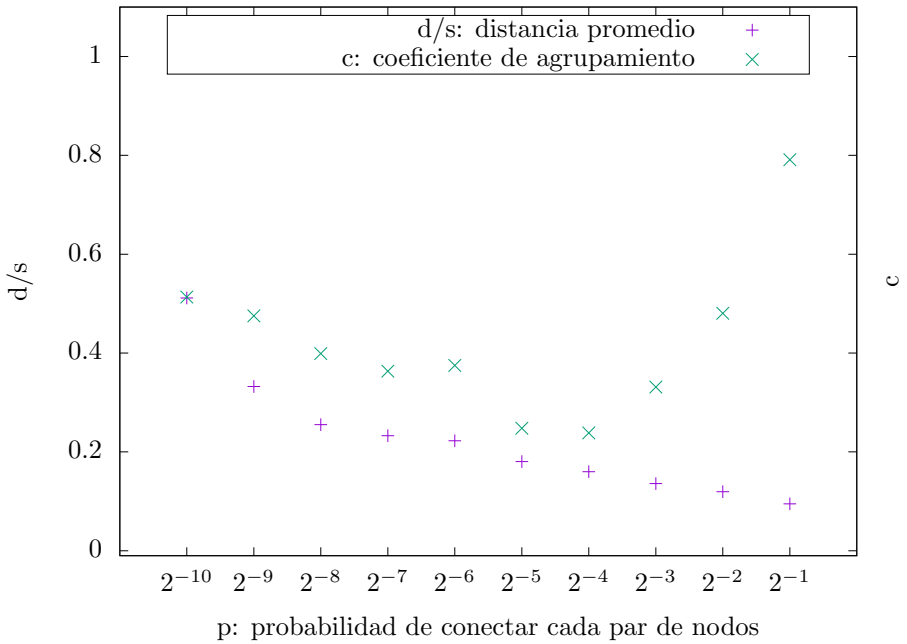


Figura 3: Resultados para k = 2

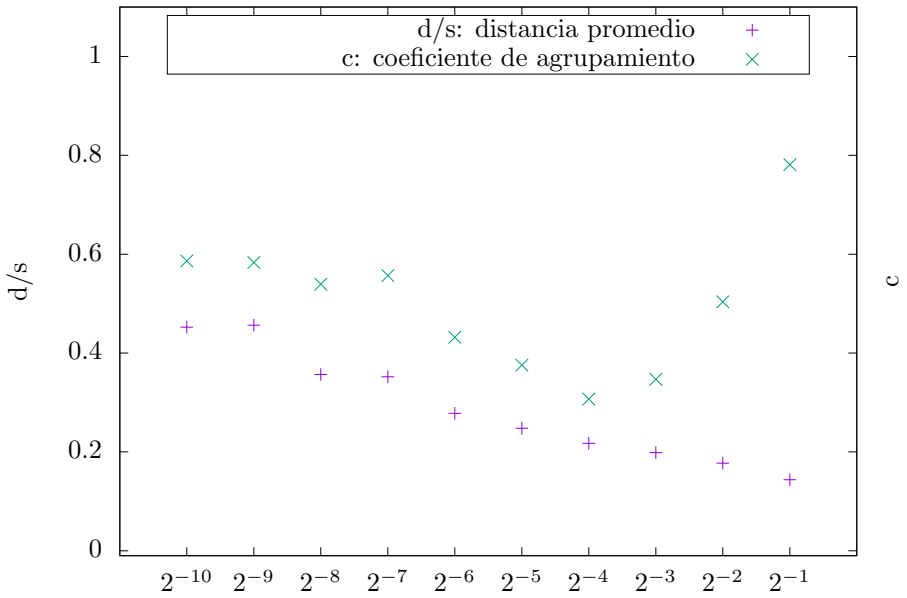


Figura 4: Resultados para k = 3

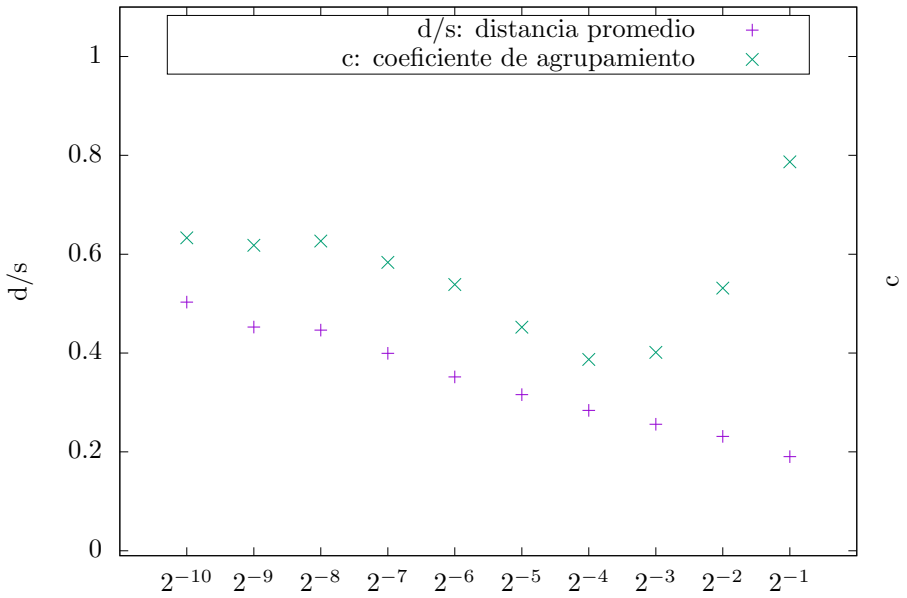


Figura 5: Resultados para k = 4

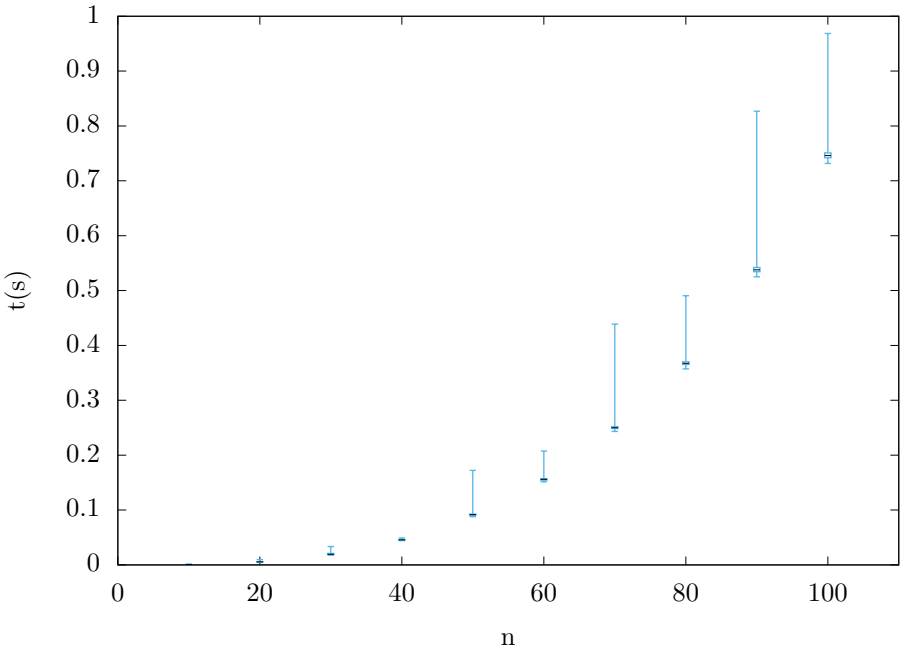


Figura 6: Resultados del Algoritmo de Floyd-Warshall

La gráfica en la figura 6 las pruebas de la complejidad computacional del algoritmo de Floyd-Warshall con este tipo de grafos. Estas pruebas se hicieron con 300 muestras de grafos aleatorios para cada tamaño de grafo. Los grafos que se generaron tienen un $k = 2$, es decir que cada nodo del mismo tiene al menos dos vecinos y el resto de los pares de nodos se conectan con una probabilidad de 2^{-2}.

4 Conclusiones

Las gráficas que comparan a las propiedades estructurales muestran que conforme aumenta la probabilidad de conexión la distancia promedio siempre se reduce progresivamente, mientras que el coeficiente de agrupamiento en general tiende a bajar para valores pequeños de p pero finalmente crece de manera casi exponencial. Por otro lado, los resultados de la complejidad computacional del algoritmo de Floyd-Warshall muestran un crecimiento consistente y progresivamente más variable conforme aumenta el número de nodos en el grafo.

References

[1] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[2] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

Tarea 5: Experimentos con el algoritmo de Ford-Fulkerson

María Gabriela Sandoval Esquivel

Mayo 2018

1 Introducción

La función del algoritmo de Ford-Fulkerson que se desarrolló en tareas anteriores da como resultado el flujo máximo que existe entre un par de nodos dado y el tiempo de ejecución del algoritmo. Este algoritmo encuentra todos los caminos que existen entre el nodo de inicio y el final y calcula el flujo máximo que se puede enviar a través de ellos. En esta tarea se hicieron pruebas de este algoritmo con un tipo especial de grafos para evaluar su desempeño conforme se eliminan nodos y aristas del grafo. La idea es analizar cómo se afecta el flujo máximo y el tiempo de ejecución del algoritmo conforme se quitan nodos y aristas del grafo hasta que el nodo inicial y el final quedan desconectados.
El tipo de grafos que se usaron para esta tarea pretenden asemejar una red social donde existen varias conexiones locales y algunas pocas conexiones a larga distancia. Para simular esta relación en el espacio se partió de un arreglo de nodos en forma de matriz cuadrada. Se definió como k el número de nodos por fila y de este modo el número total de nodos es k^2. Se definió otro parámetro l como el número de conexiones locales de cada nodo del grafo. De este modo se conectaron todo par de nodos que tuvieran una distancia Manhattan [1] entre ellos menor o igual al parámetro l. Además, a estas conexiones locales se les asignaron capacidades de flujo aleatoriamente conforme a una distribución normal. A los pares de nodos con distancia Manhattan mayor a l se les asignó una conexión de acuerdo con una probabilidad muy pequeña p para que hubiera pocas conexiones de larga distancia. A estas conexiones se les asignó una capacidad de flujo de acuerdo con una distribución exponencial.
Una vez que se construyeron los grafos se hicieron las pruebas con el algoritmo de Ford-Fulkerson teniendo como nodo inicial el de la esquina superior izquierda y nodo final el de la esquina inferior derecha. El hecho de que existan conexiones locales entre los nodos asegura que exista al menos un camino entre el nodo final y el inicial.
Las pruebas que se realizaron son de dos tipos: una donde se quitaron al azar aristas del grafo y otra donde se quitaron los nodos. La variabilidad de las pruebas depende de los parámetros k, l y p con los que se generaron varios tipos de grafos. Para cada tipo de grafo se generaron algunas muestras con las que

se analizó el desempeño del algoritmo Ford-Fulkerson conforme se percolaba el grafo hasta que el nodo inicial y final quedaran desconectados.
El motivo de estas pruebas es analizar qué tan cruciales son ciertos nodos para mantener el flujo en el grafo y cómo se comporta el flujo y los tiempos de ejecución del algoritmo de Ford-Fulkerson conforme se reduce el grafo. Otro resultado de estas pruebas es que una vez que se desconectan los nodos inicial y final se obtiene un conjunto de corte (ya sea de nodos o de aristas) con los elementos del grafo que se han quitado. La cardinalidad de este conjunto es una cota superior de la cardinalidad del mínimo conjunto de corte para los nodos inicial y final [2].

2 Programación de funciones

Para esta tarea primero se creó la función que generaba el tipo de grafos que se describió en la introducción. Esta función recibía como datos de entrada los parámetros k, l y p y asignaba las conexiones y las capacidades de flujo como se describió antes.
Después se crearon funciones para quitar aristas y nodos del grafo. Al quitar una arista, no solo se eliminaba de la lista de aristas del grafo, también era necesario quitar de la lista de vecinos del nodo de salida el nodo de llegada (y viceversa si se trataba de una arista no-dirigida). Al quitar un nodo, se eliminaba de la lista de nodos del grafo y para todos los vecinos de tal nodo se eliminaban sus aristas con la función de quitar arista. Se quitaban también las aristas dirigidas que llegaban al nodo por eliminar.

3 Pruebas experimentales

A continuación, se presentan gráficas que muestran el comportamiento del algoritmo de Ford Fulkerson conforme se removían aristas y nodos para los diferentes tipos de grafos. Para cada prueba se midió la evolución del flujo máximo del grafo y el tiempo que llevó la ejecución del algoritmo en cada iteración. Las graficas muestran las diferentes muestras que se consideraron para cada tipo de grafo con colores distintos. Para el flujo máximo se consideraron medidas absolutas con respecto al flujo obtenido inicialmente. En el eje horizontal se miden los nodos o aristas que se quitaron en cada iteración del algoritmo, en algunos casos se quitaron dos o cinco objetos de una vez para agilizar el análisis.

Las figuras 1 a 16 muestran los resultados sobre la eliminación de nodos para distintos tipos de grafos y el resto de las figuras muestran resultados para eliminación de aristas.

Las figuras 1 y 2 muestran los resultados para los grafos con 10^2 nodos, conexiones locales de 1 y probabilidad de conexiones a larga distancia de 10^{-3}.

Estas gráficas muestran los resultados sobre cinco grafos de este tipo. El flujo máximo permanece estable con algunos cambios abruptos y el tiempo de ejecución es consistentemente decreciente. El número máximo de nodos que se quitaron para tener un flujo de cero es 180.

El efecto de la probabilidad de conexiones a larga distancia se aprecia en las diferencias de estas gráficas con las que se encuentran en las figuras 3 y 4 donde esta probabilidad es de 10^{-3}. Ahí se aprecia un caso muy estable en cuanto al flujo máximo y el máximo número de nodos que se quitan para alcanzar un flujo máximo de cero es cercano a 35.

4 Conclusiones

Las gráficas que se obtuvieron de las pruebas muestran que el tiempo de ejecución del algoritmo de Ford Fulkerson siempre fue decreciente conforme se eliminaban objetos del grafo. En cuanto al flujo máximo es común encontrar gráficas escalonadas que indican que el flujo se mantiene estable hasta que se quitan ciertas aristas cruciales que muy probablemente pertenecen a un conjunto de corte de los nodos inicial y final del grafo.

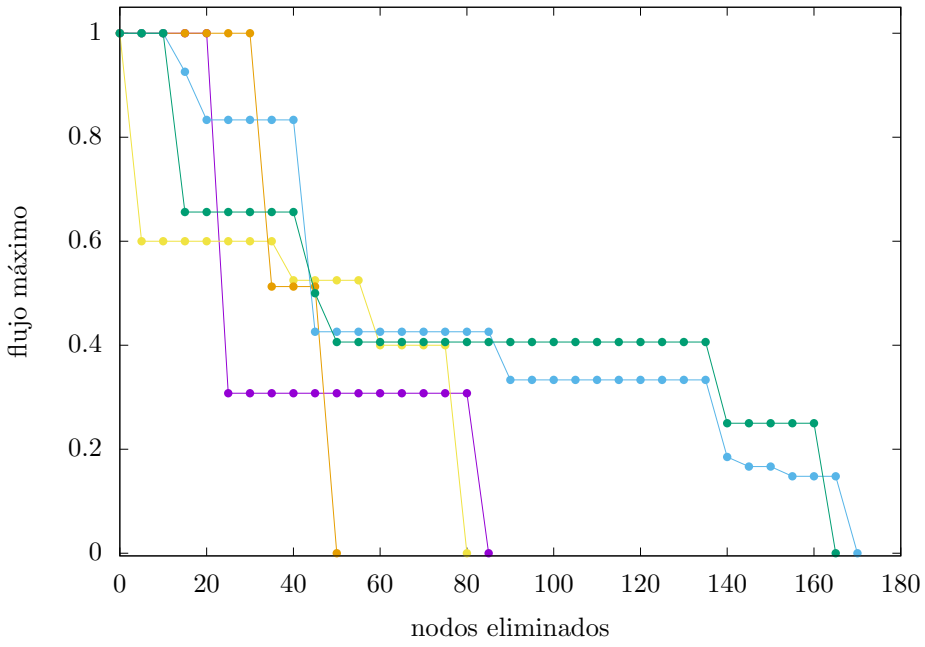


Figura 1: Flujo máximo quitando nodos para grafo con $k = 10$, $l = 1$ y $p = 10^{-3}$

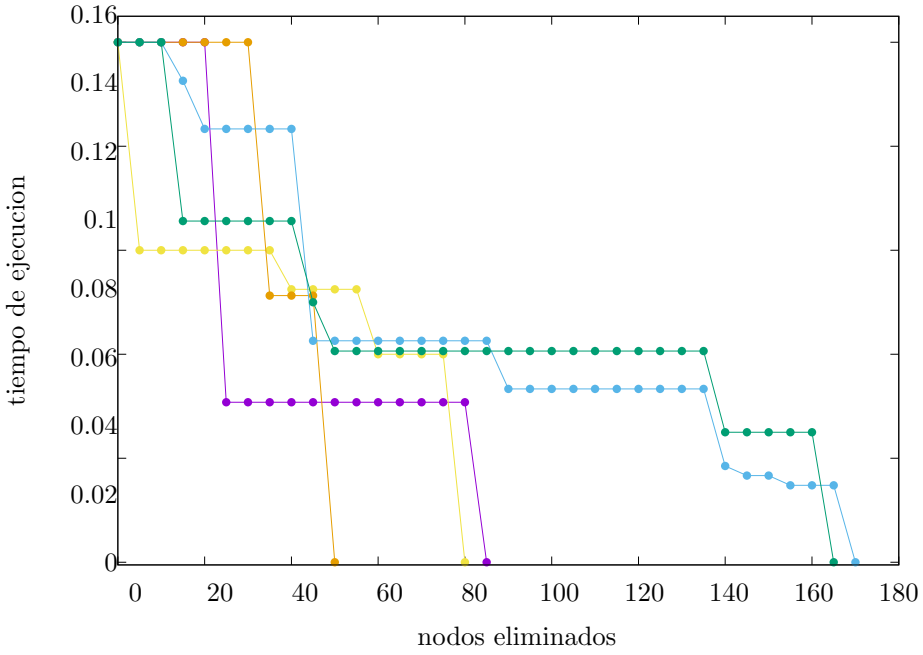


Figura 2: Tiempo de ejecución quitando nodos para grafo con $k = 10$, $l = 1$ y $p = 10^{-3}$

References

[1] Eric W Weisstein et al. Taxicab metric. *From MathWorld–A Wolfram Web Resource*, 2010.

[2] Xiao Chen, Zhe-Ming Lu, Hong-Wa Yang, and Gao-Feng Pan. A new approach to constructing a minimum edge cut set based on maximum flow.

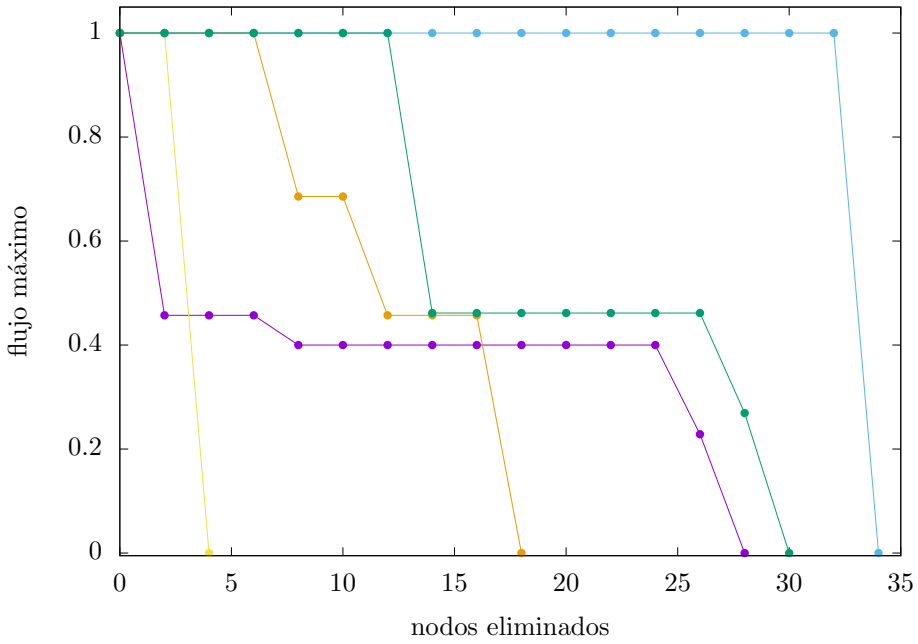


Figura 3: Flujo máximo quitando nodos para grafo con $k = 10$, $l = 1$ y $p = 10^{-4}$

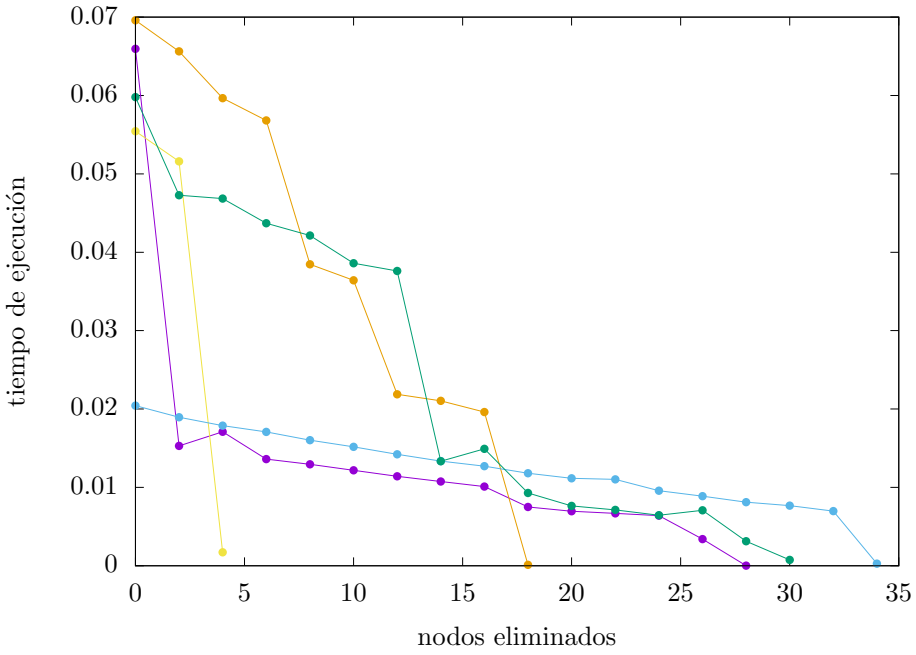


Figura 4: Tiempo de ejecución quitando nodos para grafo con $k = 10$, $l = 1$ y $p = 10^{-4}$

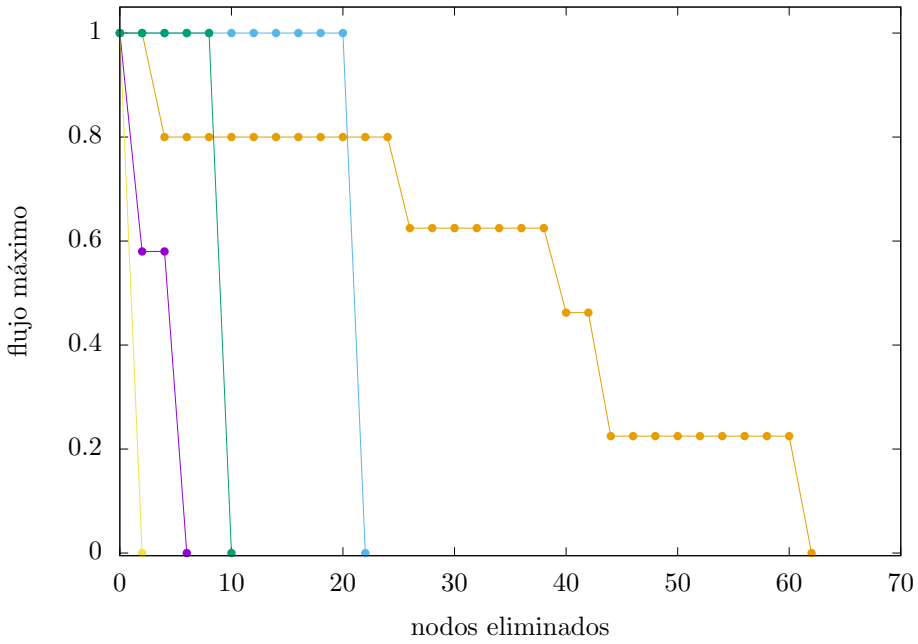


Figura 5: Flujo máximo quitando nodos para grafo con $k = 10$, $l = 2$ y $p = 10^{-4}$

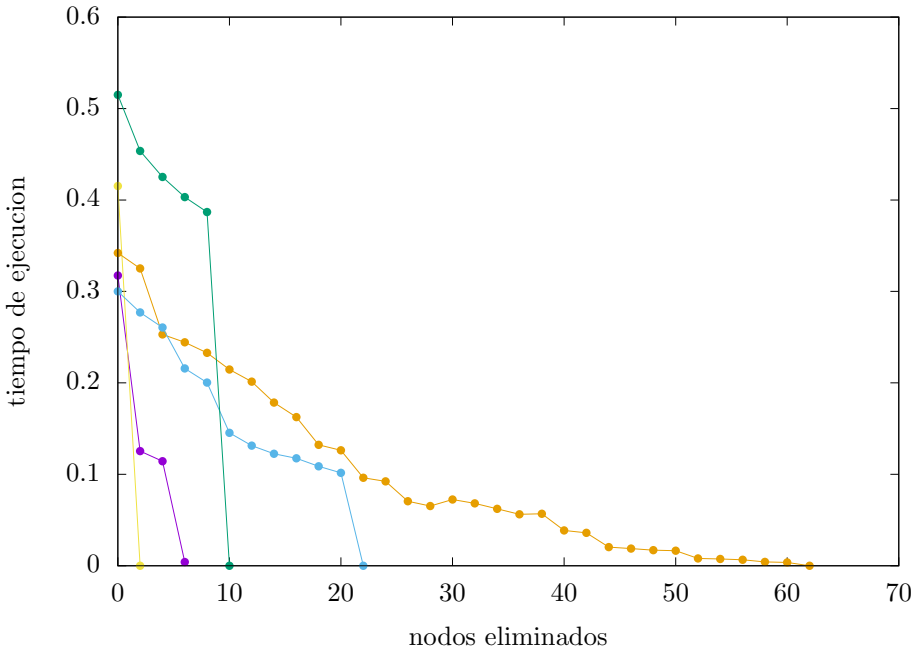


Figura 6: Tiempo de ejecución quitando nodos para grafo con $k = 10$, $l = 2$ y $p = 10^{-4}$

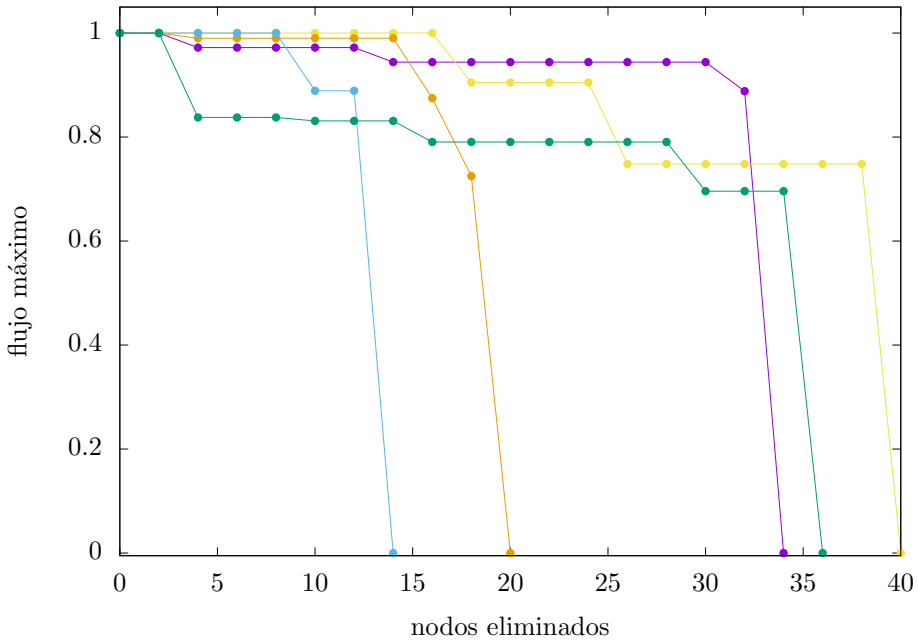


Figura 7: Flujo máximo quitando nodos para grafo con $k = 10$, $l = 3$ y $p = 10^{-4}$

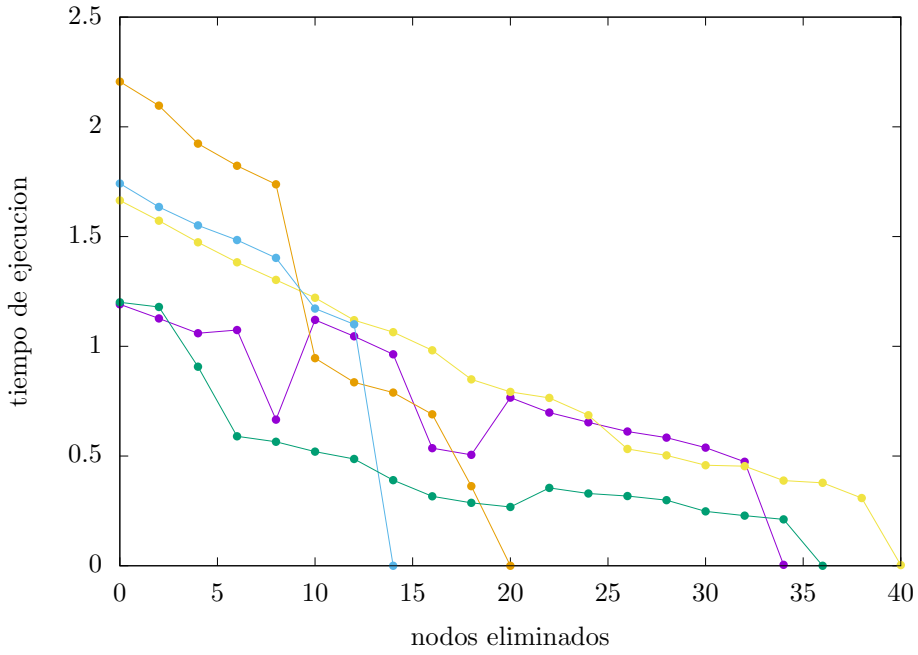


Figura 8: Tiempo de ejecución quitando nodos para grafo con $k = 10$, $l = 3$ y $p = 10^{-4}$

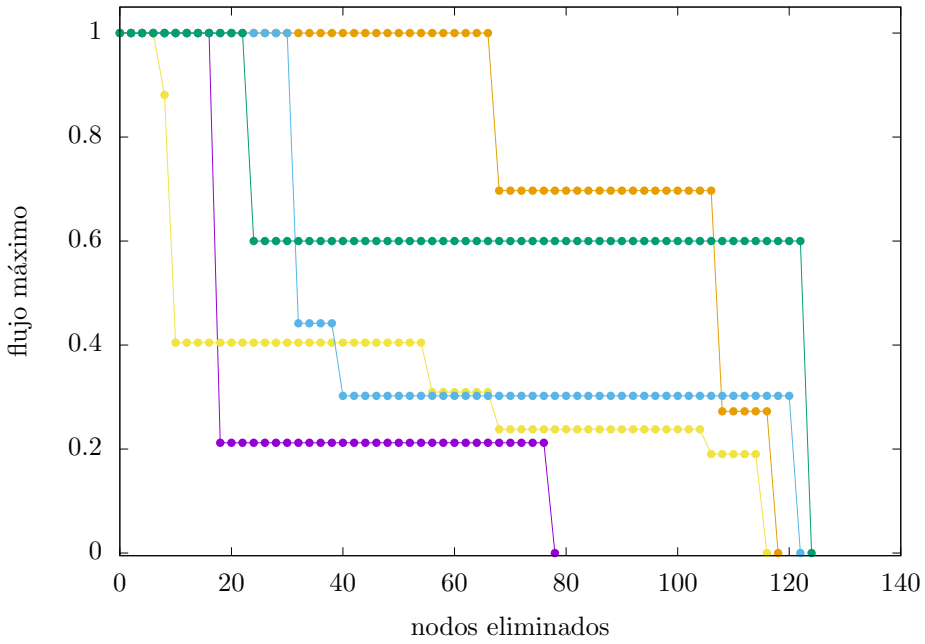


Figura 9: Flujo máximo quitando nodos para grafo con $k = 20$, $l = 1$ y $p = 10^{-4}$

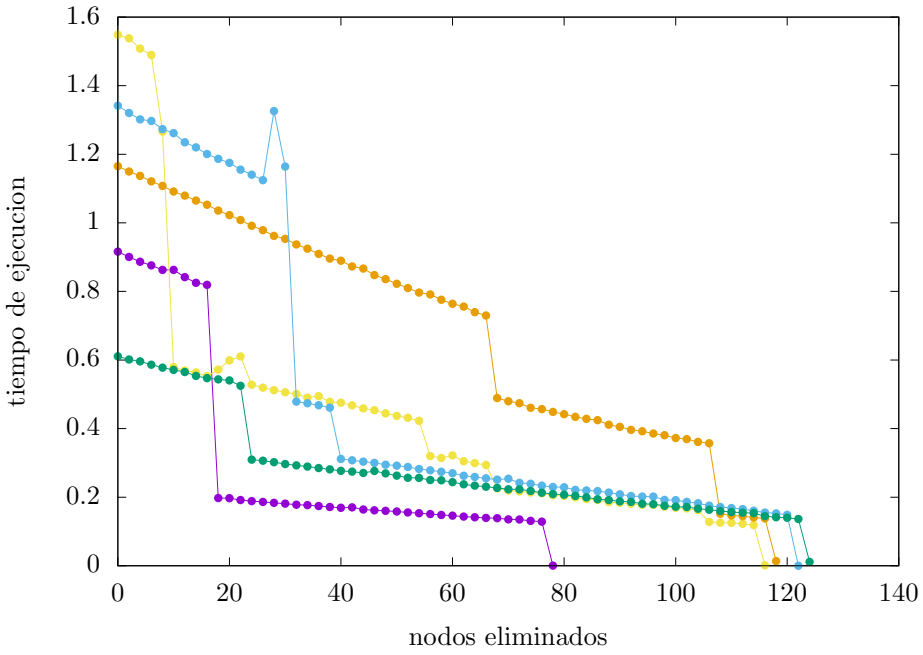


Figura 10: Tiempo de ejecución quitando nodos para grafo con $k = 20$, $l = 1$ y $p = 10^{-4}$

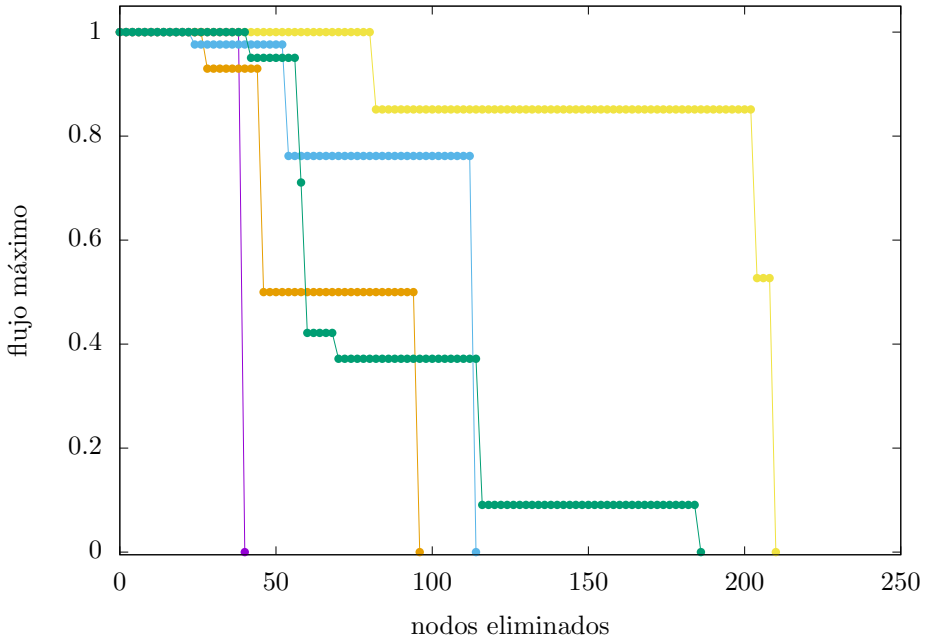


Figura 11: Flujo máximo quitando nodos para grafo con $k = 20$, $l = 2$ y $p = 10^{-4}$

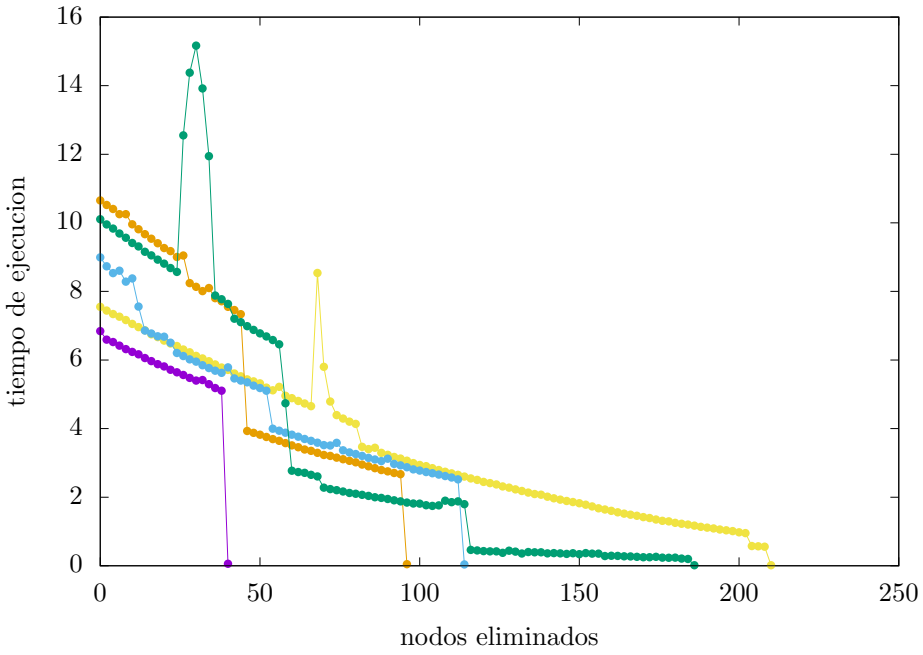


Figura 12: Tiempo de ejecución quitando nodos para grafo con $k = 20$, $l = 2$ y $p = 10^{-4}$

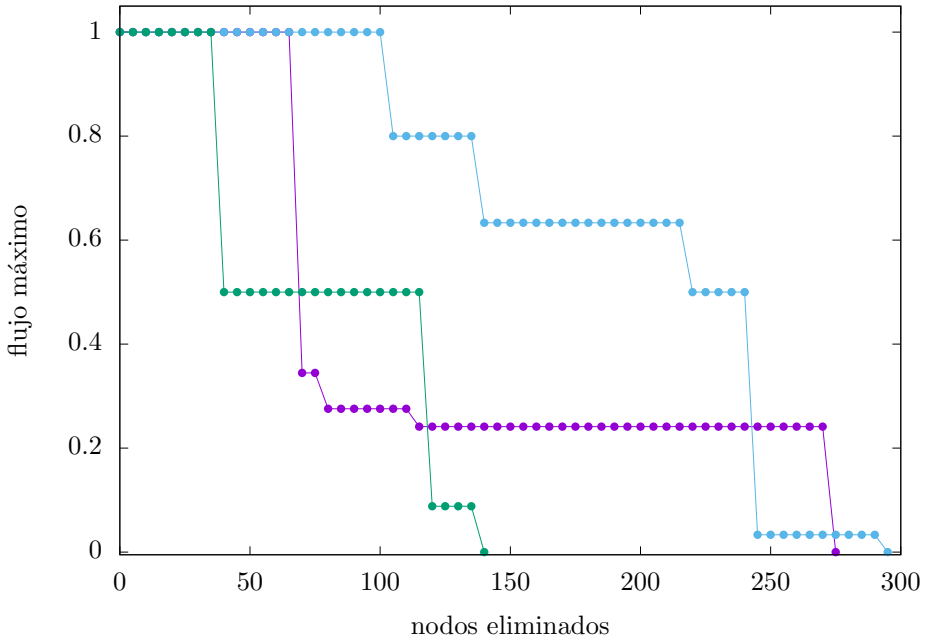


Figura 13: Flujo máximo quitando nodos para grafo con $k = 30$, $l = 1$ y $p = 10^{-4}$

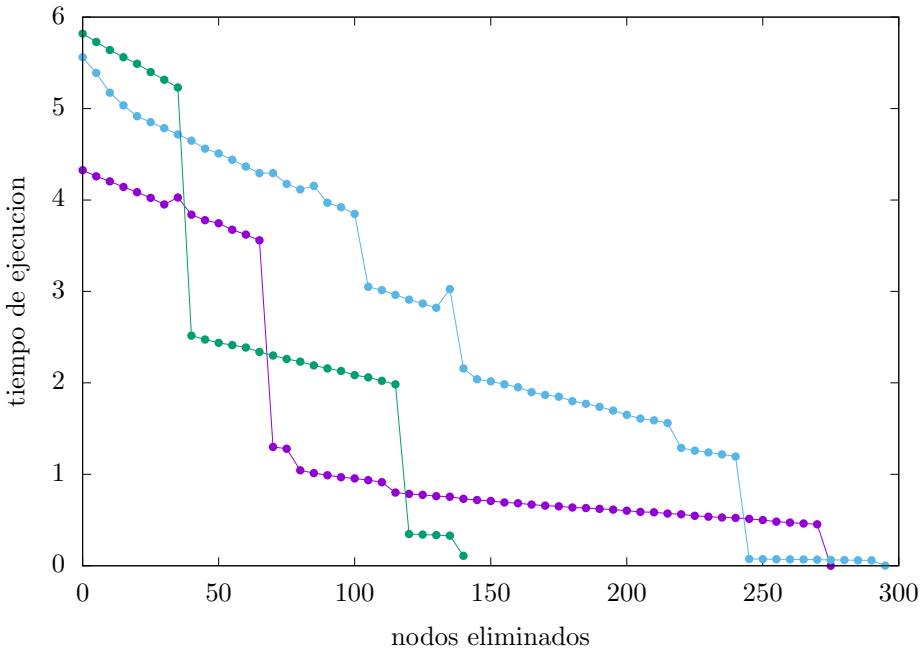


Figura 14: Tiempo de ejecución quitando nodos para grafo con $k = 30$, $l = 1$ y $p = 10^{-4}$

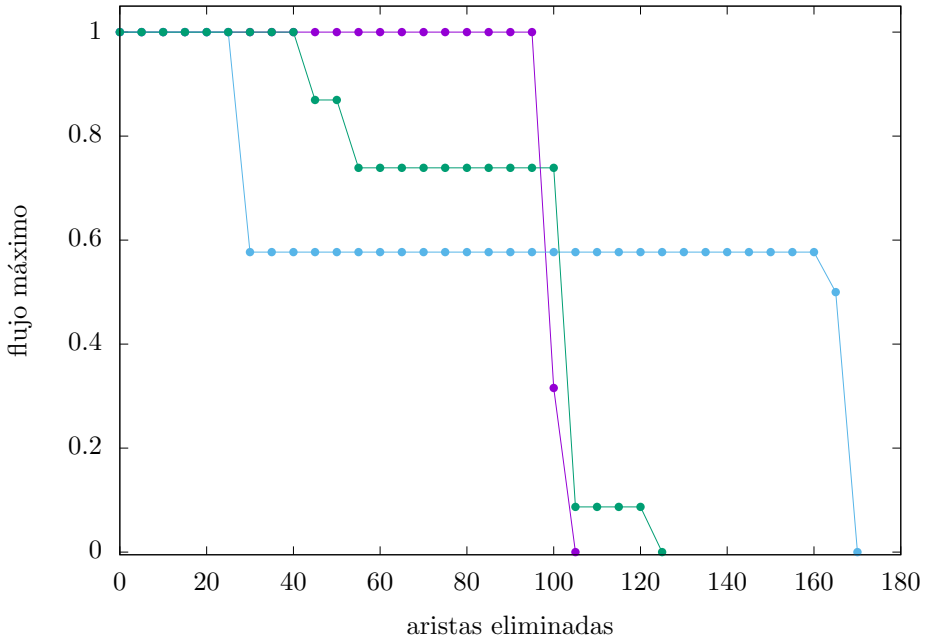


Figura 15: Flujo máximo quitando aristas para grafo con $k = 10$, $l = 1$ y $p = 10^{-4}$

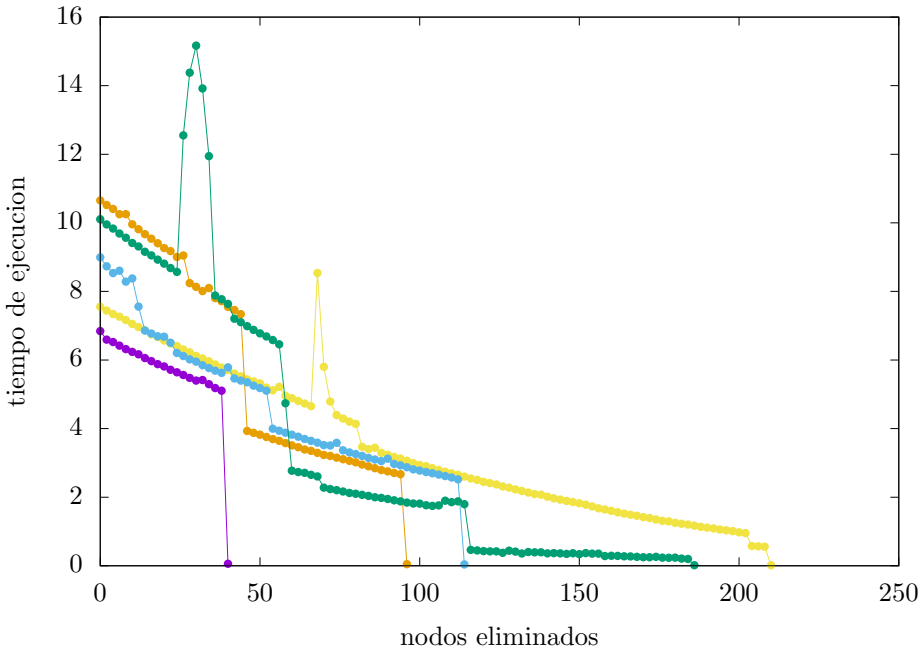


Figura 16: Tiempo de ejecución quitando aristas para grafo con $k = 10$, $l = 1$ y $p = 10^{-4}$

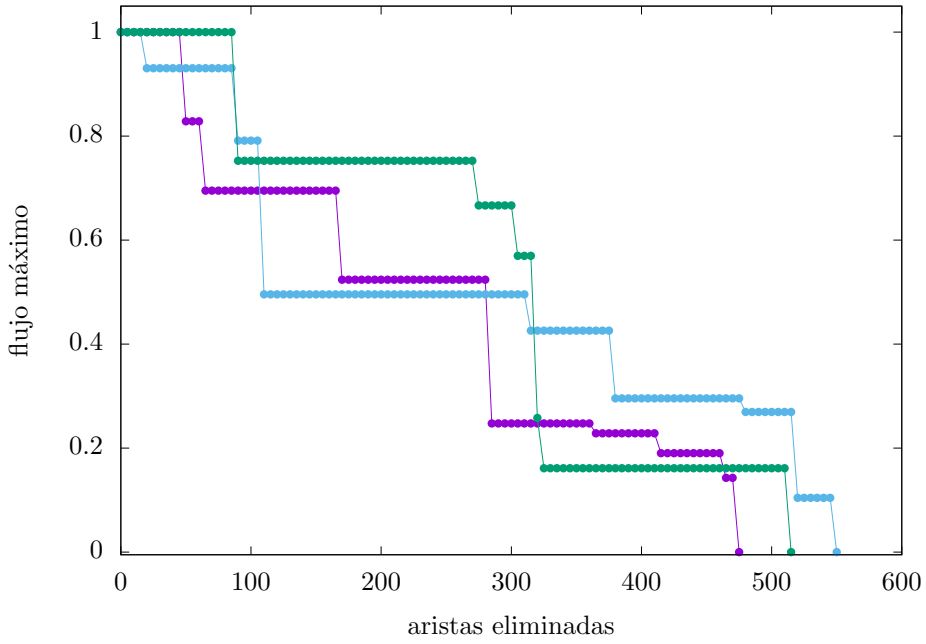


Figura 17: Flujo máximo quitando aristas para grafo con $k = 10$, $l = 2$ y $p = 10^{-4}$

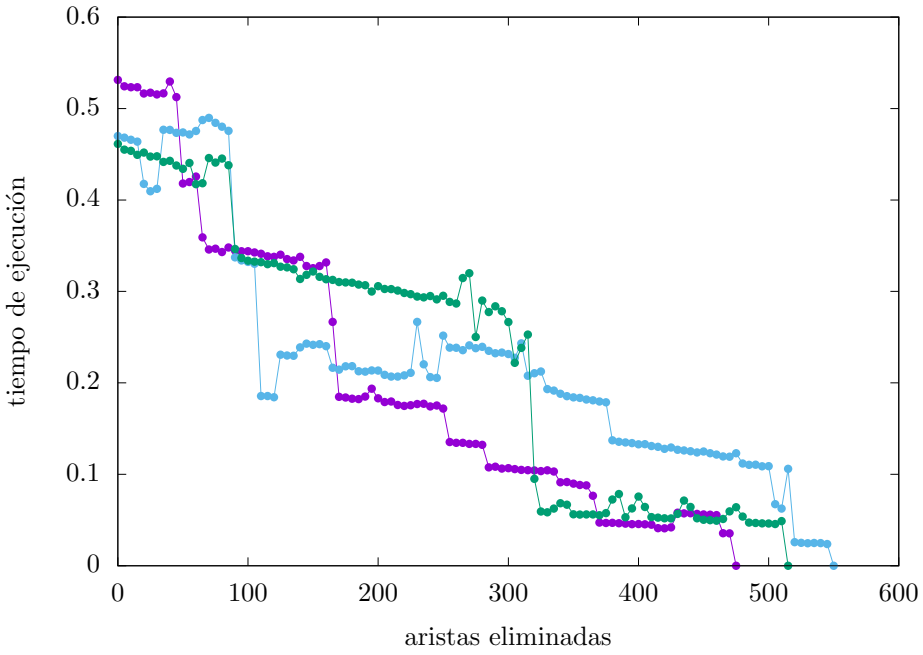


Figura 18: Tiempo de ejecución quitando aristas para grafo con $k = 10$, $l = 2$ y $p = 10^{-4}$

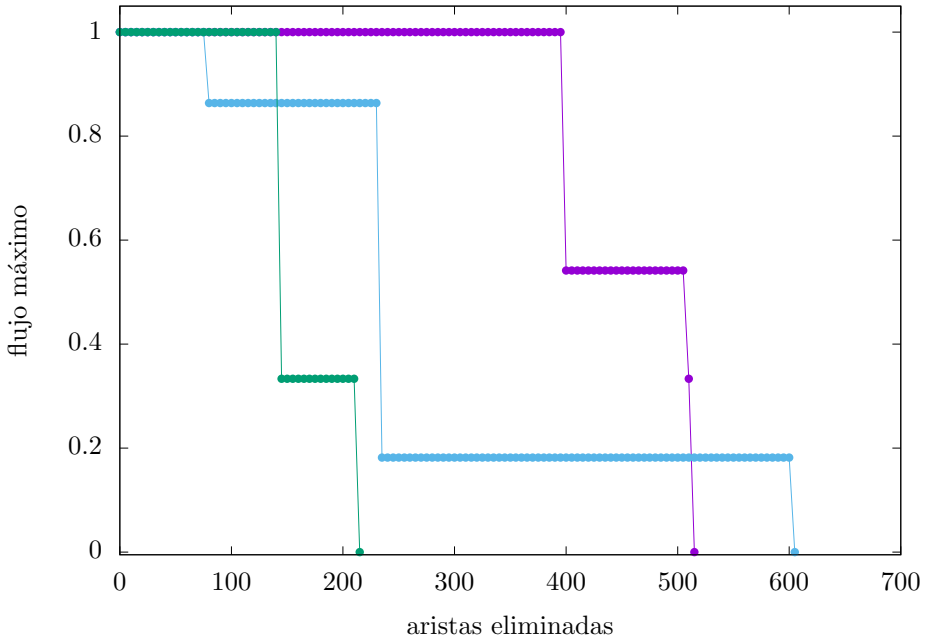


Figura 19: Flujo máximo quitando aristas para grafo con $k = 20$, $l = 1$ y $p = 10^{-4}$

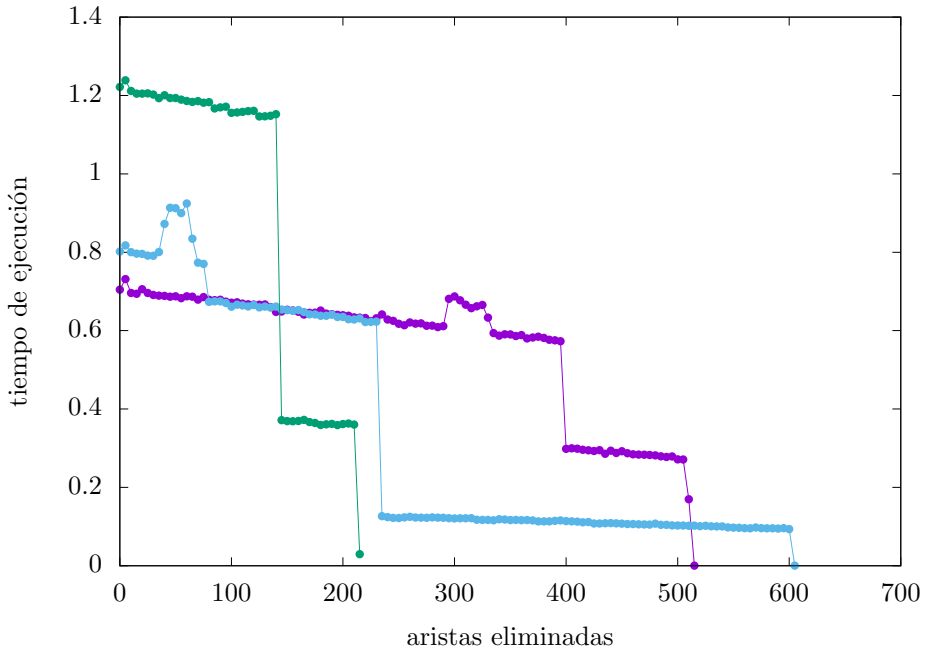


Figura 20: Tiempo de ejecución quitando aristas para grafo con $k = 20$, $l = 1$ y $p = 10^{-4}$

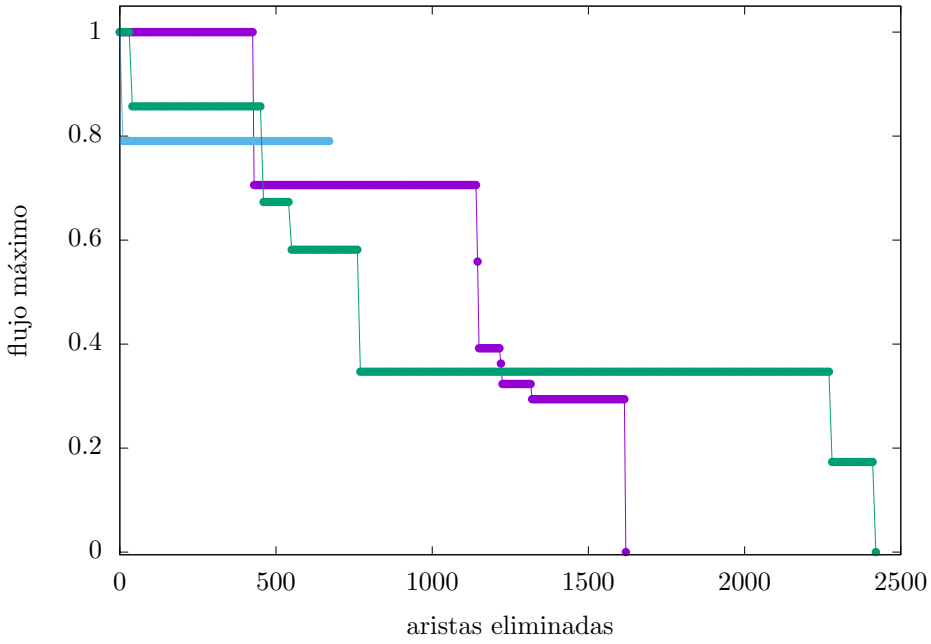


Figura 21: Flujo máximo quitando aristas para grafo con $k = 20$, $l = 2$ y $p = 10^{-4}$

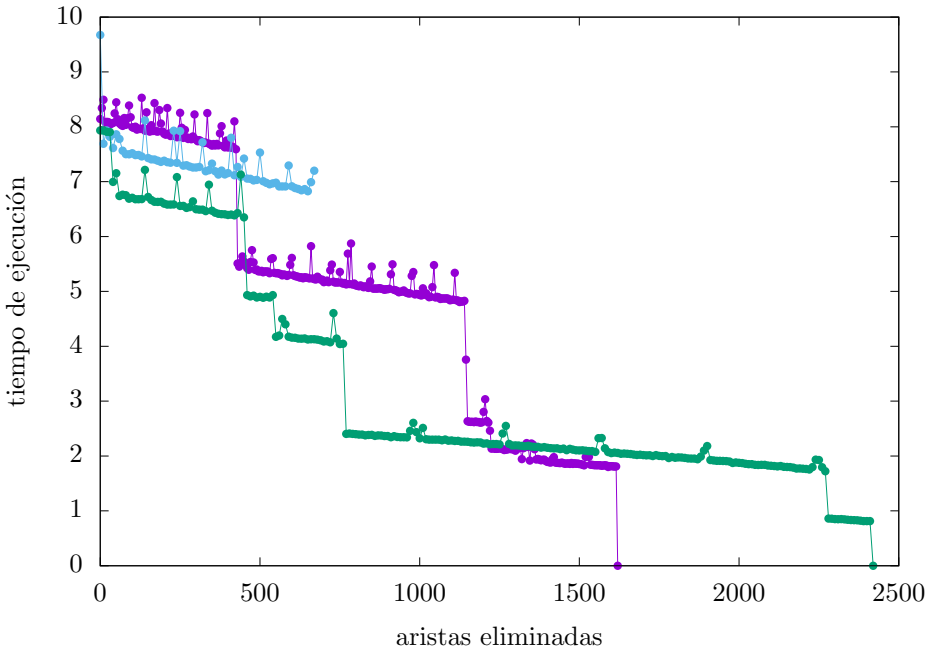


Figura 22: Tiempo de ejecución quitando aristas para grafo con $k = 20$, $l = 2$ y $p = 10^{-4}$

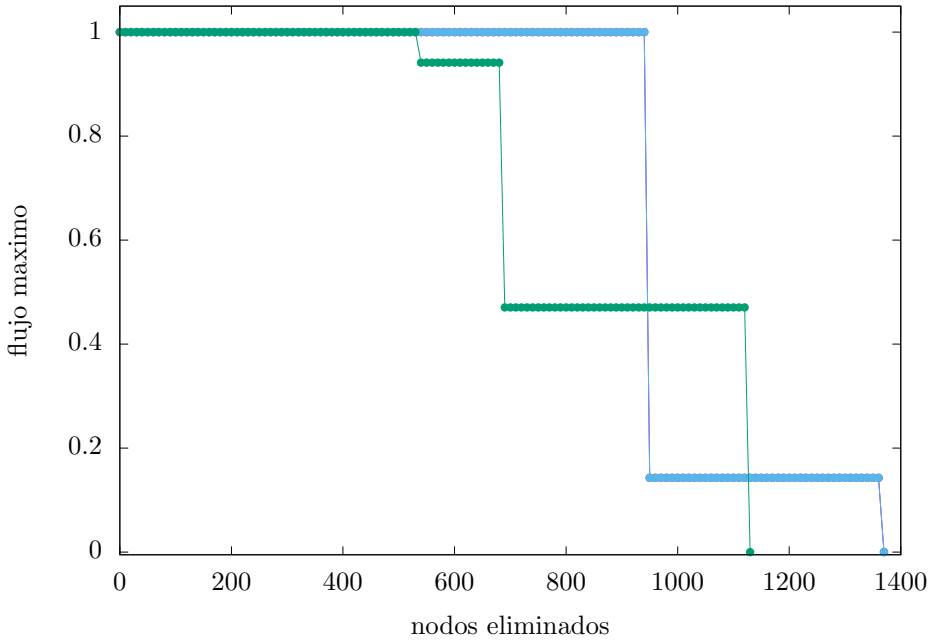


Figura 23: Flujo máximo quitando aristas para grafo con $k = 30$, $l = 1$ y $p = 10^{-4}$

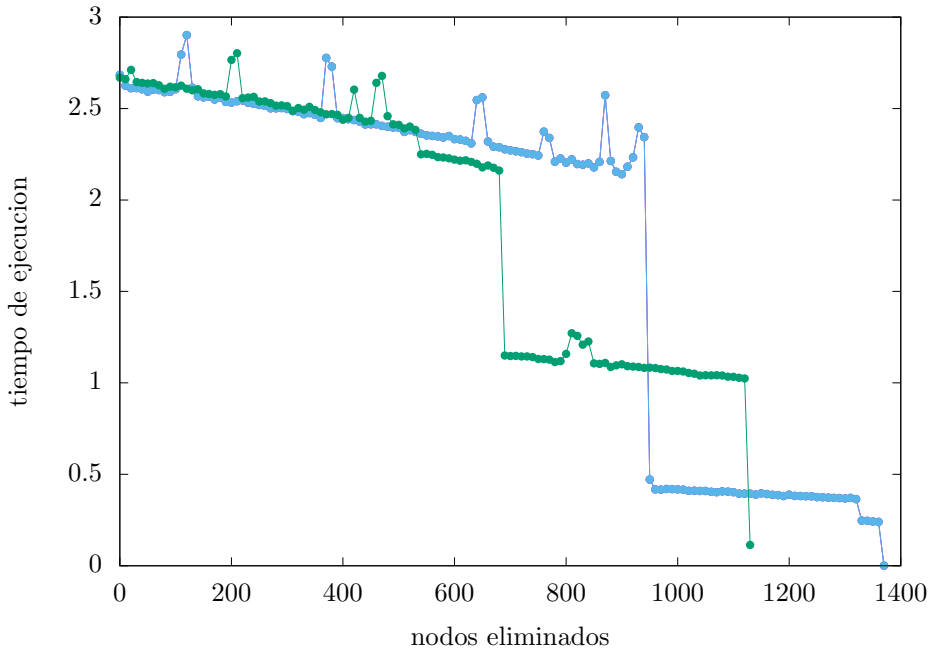


Figura 24: Tiempo de ejecución quitando aristas para grafo con $k = 30$, $l = 1$ y $p = 10^{-4}$

Tarea 6: Experimentos con el algoritmo aproximado de contracción para el corte mínimo

María Gabriela Sandoval Esquivel

Mayo 2018

1 Introducción

El algoritmo aproximado de contracción para corte mínimo fue presentado por David R. Krager en 1996 [1]. Su desarrollo se enfoca en el problema de corte mínimo general en el que se busca encontrar el mínimo corte que divida a un grafo conectado a la mitad (o alternativamente en r sub-partes). El algoritmo consiste en un proceso iterativo en el cual, en cada corrida, se contrae sucesivamente el grafo hasta que queda reducido a un simple grafo con dos nodos y una arista entre ellos que representa una cota superior al corte mínimo. El proceso culmina cuando se encuentra la mínima de estas cotas. El proceso de contracción sucesiva del grafo consiste en elegir dos nodos al azar, que estén conectados, para contraerlos en un solo nodo que tiene como vecinos a la unión de los vecinos de los nodos que los forman. La sustentación de la efectividad de este algoritmo se basa en el hecho de que el corte mínimo regularmente está formado por relativamente pocos nodos del grafo y por lo tanto la probabilidad de elegir una arista que forme parte del corte mínimo es baja.
En esta tarea se presenta una adaptación de este algoritmo para el problema de corte mínimo s-t. Este problema se diferencia del general en el sentido en el que se tienen dos nodos s y t que necesitan quedar separados en el corte. Este problema tiene cierta dualidad con el problema de flujo máximo s-t que se resuelve con el algoritmo de Floyd-Fulkerson que se desarrolló en tareas pasadas. En este reporte se presenta la adaptación del algoritmo de Krager para este problema y pruebas que comparan la efectividad de este método de aproximación con el algoritmo de Floyd-Fulkerson que se programó en tareas pasadas.

2 Programación de funciones

Para esta tarea se programó una función para realizar la contracción sucesiva de un grafo dado de tal modo que dos nodos dados quedaran en partes separadas y que devolviera el valor del corte resultante. Tal función elegía al azar dos nodos del grafo y los contraía, formando un nuevo nodo que tenía como vecinos a la unión de vecinos de los nodos que lo formaban. La arista entre dos nodos

Table 1: Comparación del desempeño de los algoritmos (FF: Floyd Fulkerson, A: Adaptación del algoritmo de contracción.

Instancia (k,l)	Diferencia de Tiempo ((A-FF)/A)	Gap de corte mínimo (A-FF
(10,1)	39.062	0
(10,2)	31.336	0
(10,3)	12.068	0
(20,1)	1.682	0
(20,2)	1.291	0
(30,1)	2.917	0
(30,2)	37.58	0
(40,1)	103.25	2.667
(40,2)	28.678	34.14286

contraídos se eliminaba de la lista de aristas y el nuevo nodo tenía como nombre la unión de los nombres de los nodos que lo formaban. Para que dos nodos se pudieran contraer era importante verificar que no se unieran los nodos que se especificaron en un principio. Para esto, se agregó un atributo a los nodos que representaban su estatus, es decir si un nodo contenía a alguno de los nodos especificado o no. Este atributo tenía como valor inicial de cero para todos los nodos del grafo original, excepto los nodos s y t que tenían valores de 2 y 3 específicamente. Para verificar si dos nodos podían contraerse se calculaba la diferencia entre sus valores de estatus, solo si ésta era diferente a 1 los nodos podían contraerse. Una vez contraídos, el nuevo nodo heredaba el estatus de mayor valor de los nodos que lo formaron.
El algoritmo hace uso de esta función para el número de iteraciones especificado y guarda la mejor cota superior para el corte mínimo s-t en cada iteración.

3 Pruebas experimentales

A continuación, se presentan gráficas que muestran el proceso para encontrar la cota superior para el corte mínimo s-t con el algoritmo que se desarrolló. El tipo de grafos que se usaron para esta tarea son los mismos que se usaron en la tarea anterior. Son grafos formados por un enrejado de nodos de kxk en el que todos los nodos a una distancia manhattan de al menos l están conectados y existen conexiones a larga distancia con una probabilidad de $p = 0.001$. La solución óptima obtenida se muestra con la línea negra en cada gráfica.

4 Conclusiones

Las gráficas muetran que el proceso iterativo sí hace que las cota superiores convergan al corte mínimo que se obtiene con la solución obtenida por el algoritmo de Floyd Fulkerson. Sin embargo, la tabla 1 muestra que el algoritmo siempre es

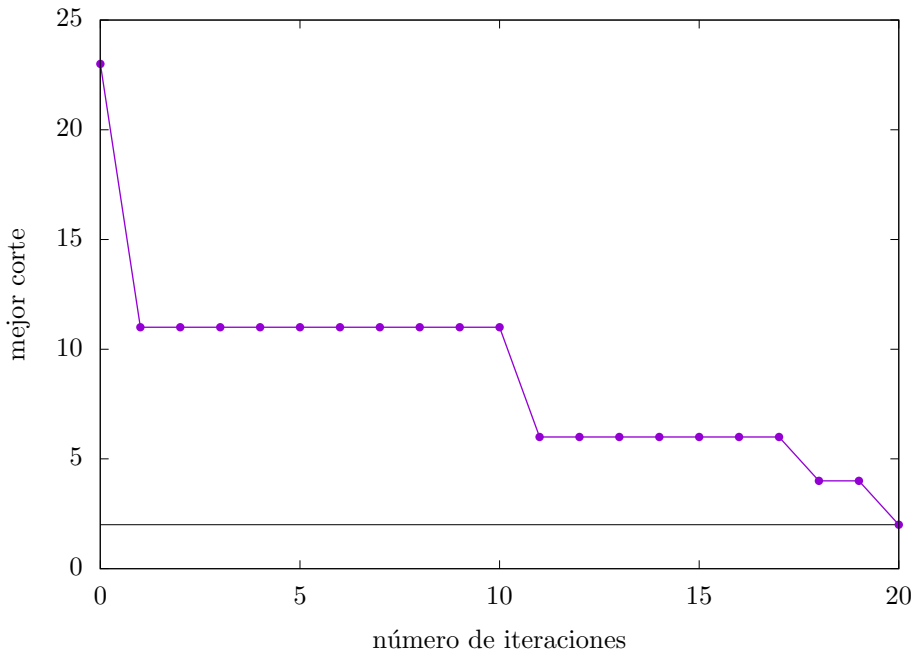


Figura 1: Resultados para k = 10, l = 1

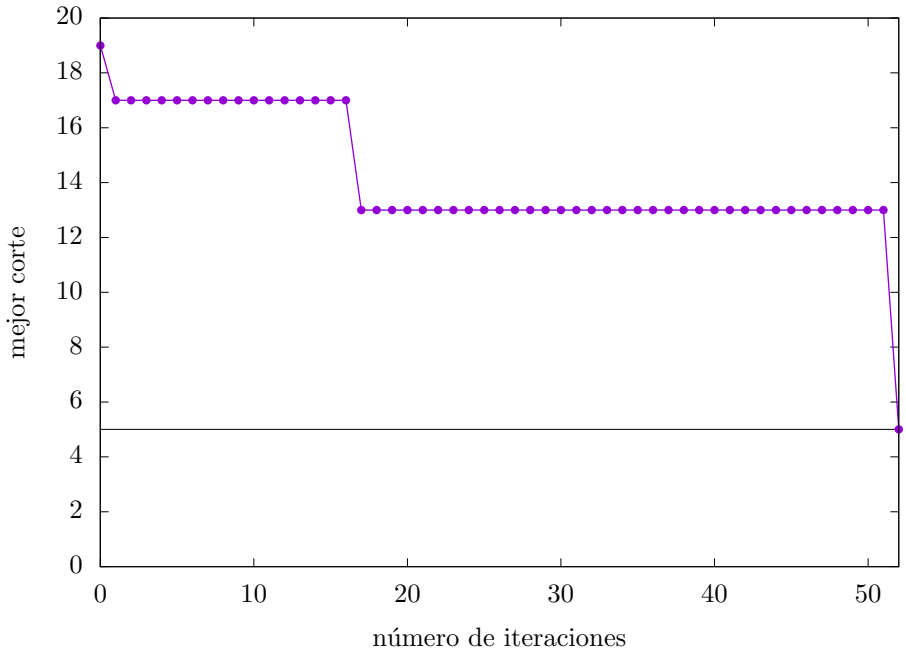


Figura 2: Resultados para k = 10, l = 2

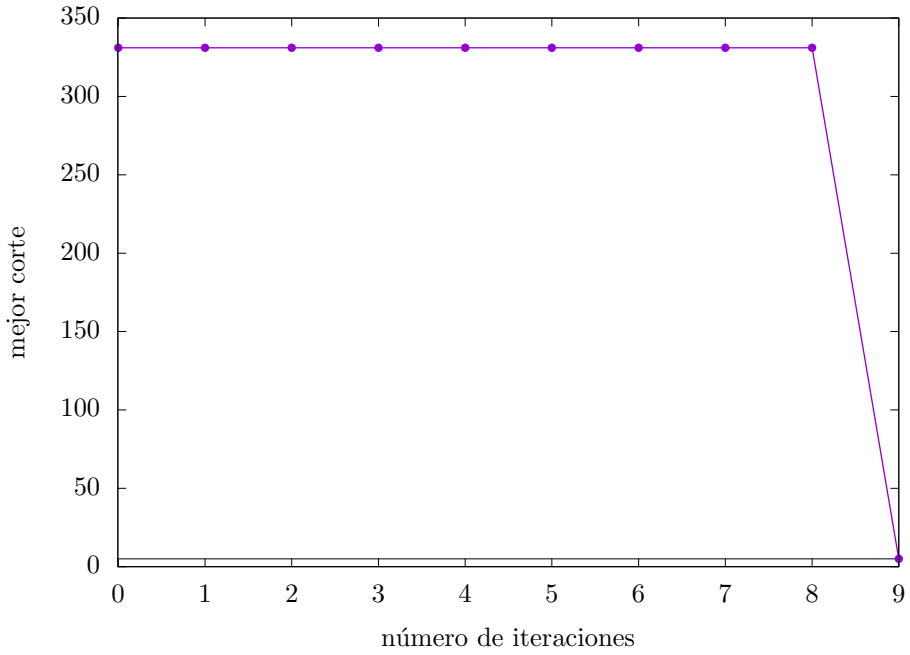


Figura 3: Resultados para k = 20, l = 2

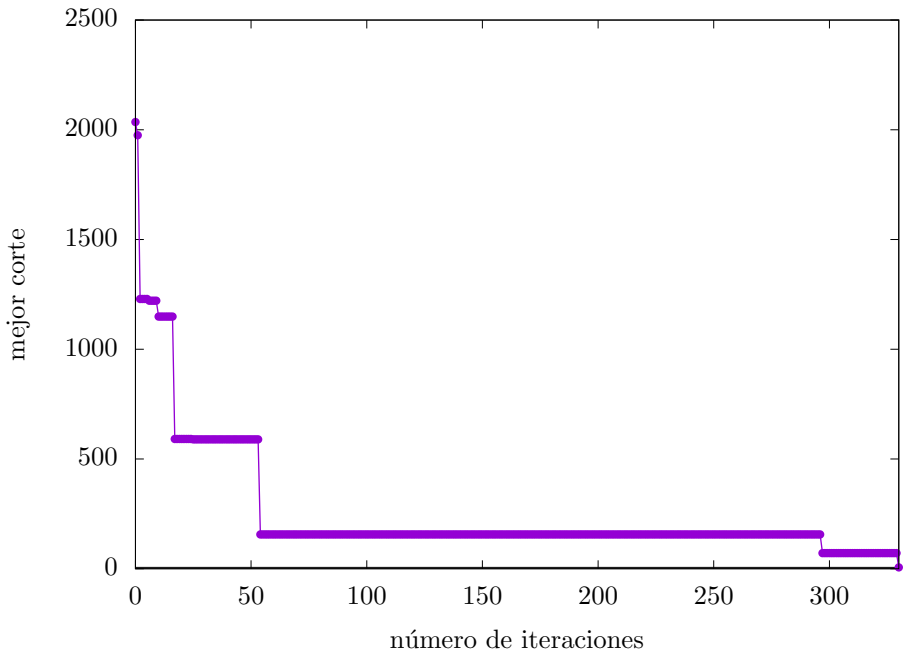


Figura 4: Resultados para k = 30, l = 2

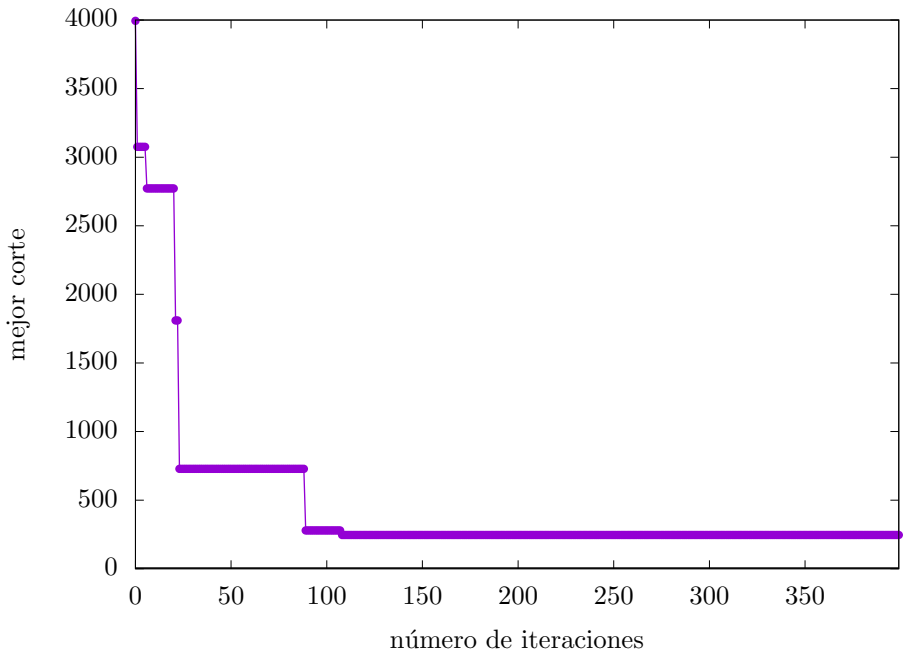


Figura 5: Resultados para k = 40, l = 2

más tardado que el Ford-Fulkerson lo que en estos casos descarta su eficiencia.

References

[1] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.