DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Optimización Algorítmica

4 de octubre de 2013 — Algoritmos y Estructuras de Datos III

Grupo En hexa

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Fernández Wortman, Maximiliano	892/10	maxifwortman@gmail.com
Gasco, Emilio	171/12	gascoe@gmail.com
Di Lorenzo, Leandro	101/06	leandro.jdl@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1 Introducción

Objetivos

El presenta trabajo pretende resolver problemas de optimización combinatoria. Se utilizaron técnicas basadas en programación dinámica, recorridos de grafos y búsquedas de árboles generadores mínimos.

Metodología a implementar

Para la realización de este trabajo utilizamos las siguientes herramientas:

- `C++` como lenguaje de programación,
- compilado bajo `gcc`,
- `matplotlib` y `gnuplot` para armar los gráficos,
- LaTeX para la composición del documento,
- bajo los entornos
  - Debian GNU/Linux
  - Ubuntu
  - FreeBSD

2 Problema 1: Impresiones ordenadas

2.1 Descripción del problema

Presentación de las características del ejercicio

En este ejercicio se nos solicita afrontar la tarea de distribución de trabajos de una imprenta. Nos indican que se dispone solamente de dos impresoras de manera que cada uno de los trabajos se realizará necesariamente en alguna de estas.

Dichas impresiones, además, sólo podrán realizarse respetando el orden en que fueron encargadas. Por último se nos hace saber que, debido a cuestiones técnicas, el costo que supone realizar la impresión de un determinado trabajo en alguna de las impresoras estará regido por el último trabajo para el que se haya utilizado la misma.

Nuestro objetivo es el de minimizar el costo de impresión de los trabajos, sabiendo de antemano la cantidad total de trabajos y contando con la lista de costos dispuesta bajo un formato específico, cuya composición será mencionada en los párrafos subsiguientes.

Distribución exclusiva de los trabajos

A partir de las características mencionadas anteriormente podemos afirmar que para todo trabajo T_k, en donde k representa el número de orden en que éste nos fue encargado, existirá la posibilidad de imprimirlos, o bien en la impresora i_1, o bien en la impresora i_2.

Si notamos I_k como la lista que contiene a todos los trabajos que fueron realizados en la impresora i_k, y T como el conjunto de todos los trabajos encargados, entendemos esta particularidad como

$$(\forall\ t_i \in T)\ t_i \in I_1 \oplus t_i \in I_2$$

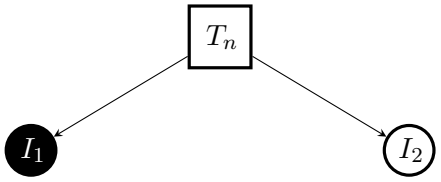

Figura 1: *Cada trabajo generará dos posibles soluciones.*

Crecimiento exponencial del universo de soluciones

La distribución exclusiva de los trabajos entre dos posibles impresoras se traduce en la existencia de un universo de soluciones cuyo crecimiento resulta exponencial (2^n) respecto a la cantidad total de trabajos encargados (n), tal y como se muestra en la *Figura 2*, en donde

los nodos negros representan a los trabajos impresos en la impresora I_1, y los nodos blancos representan a los trabajos impresos en la impresora I_2.

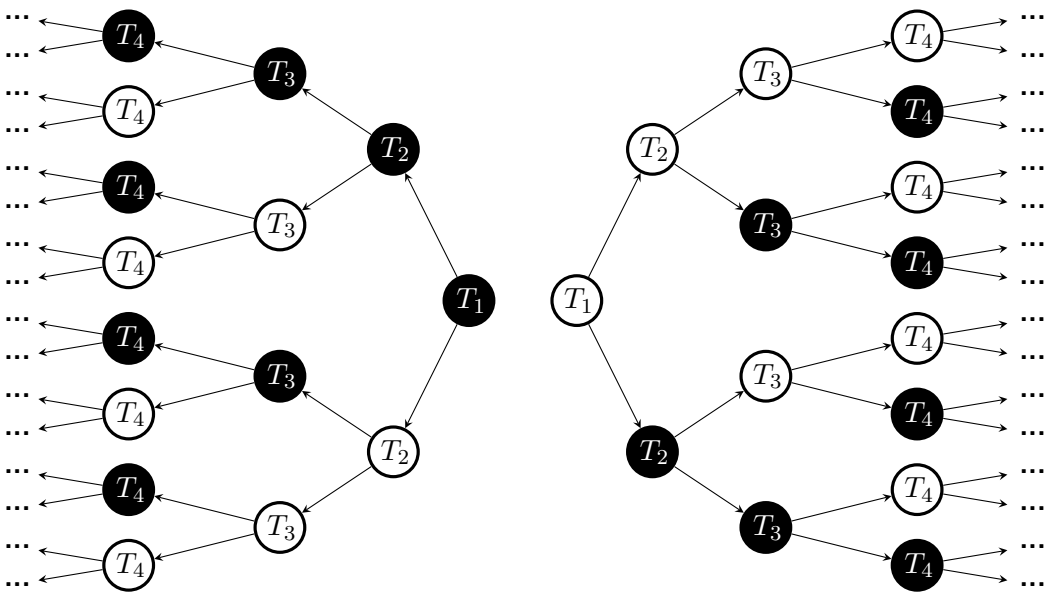

Figura 2: *Distribución de los trabajos*

Ejemplo. Teniendo en cuenta las premisas anteriores, para el caso en que se tiene una cantidad total de 3 trabajos, las posibilidades de impresión serían $2^3 = 8$.

$I_1 = [t_1, t_2, t_3]$	$I_2 = [\,]$
$I_1 = [t_1, t_2]$	$I_2 = [t_3]$
$I_1 = [t_1, t_3]$	$I_2 = [t_2]$
$I_1 = [t_1]$	$I_2 = [t_2, t_3]$
$I_1 = [\,]$	$I_2 = [t_1, t_2, t_3]$
$I_1 = [t_3]$	$I_2 = [t_1, t_2]$
$I_1 = [t_2]$	$I_2 = [t_1, t_3]$
$I_1 = [t_2, t_3]$	$I_2 = [t_1]$

respetando en cada caso

$$(\forall\ t_i, t_j \in I_{1|2})\ i < j \Rightarrow \mathsf{pos}(t_i) < \mathsf{pos}(t_j)$$

Observación. En este punto, creemos importante mencionar que es notorio el hecho de que resulta trivial distinguir la impresora en la que se realizará la impresión del *primer trabajo* (t_1) ya que, por las características del problema, teniendo en cuenta que las dos impresoras son iguales, se genera una simetría en el universo de soluciones, de forma tal que si la misma se realiza en I_1 esto generará posteriormente las mismas combinaciones que si se hubiese

realizado en I_2.

Descripción de los costos de impresión

Debido a que cada trabajo tiene un costo de impresión distinto según los trabajos que se hayan realizado anteriormente en la impresora utilizada, notaremos $c_{x,y}$ como el precio de imprimir el trabajo t_y luego de haber realizado la impresión de t_x. Particularmente, el subíndice 0 indicará que no se ha realizado todavía ningún trabajo en esa impresora, lo cual sin embargo no quita que de todas formas la impresión del trabajo tenga un costo.

$$(\forall t_x, t_y \in i, i = I_{1|2}, x < y)\ t_y = i[n] \land c_{x,y} = \{costo\ de\ t_y\} \Rightarrow t_x = i[n-1]$$

Ejemplo. Manteniendo la cantidad de 3 trabajos ya utilizada anteriormente, e implementando la notación C_n como «la lista de todos los costos potenciales que corresponden a la impresión del trabajo t_n», podemos representar los *posibles* costos para cada trabajo de la siguiente forma

$$\begin{aligned} C_1 &= [c_{0,1}] \\ C_2 &= [c_{0,2},\ c_{1,2}] \\ C_3 &= [c_{0,3},\ c_{1,3},\ c_{2,3}] \end{aligned}$$

La elección de la realización de trabajo en la impresoras I_1 o I_2 va a estar dado por la minimización del costo total. La suma de las impresiones en ambas impresoras tiene que ser mínimo con respecto a cualquier otra posible combinación de impresiones. Esto genera que para n cantidad de trabajos existen 2^n combinaciones posibles. Como ambas impresoras son indistinguibles en cuanto a los costos de impresión, la elección de la impresora con respecto al primer trabajo genera las mismas combinaciones pero intercambiando trabajos. Si se decide imprimir t_1 en I_2 las posibilidades son

$$\begin{aligned} I_1 &= \{\} & I_2 &= \{t_1, t_2, t_3\} \\ I_1 &= \{t_3\} & I_2 &= \{t_1, t_2\} \\ I_1 &= \{t_2\} & I_2 &= \{t_1, t_3\} \\ I_1 &= \{t_2, t_3\} & I_2 &= \{t_1\} \end{aligned}$$

generando los mismos posibles costos que si se imprimiese t_1 en I_1. Bajo este concepto es que se puede definir que t_1 siempre se imprime en I_1 lo cual genera una baja en la cantidad de combinaciones, siendo 2^{n-1} posibilidades para n trabajos. Se pide resolver el problema con complejidad temporal en peor caso de $\mathcal{O}(n^2)$

2.2 Hipótesis de resolución

Se imprimen n trabajos en orden en dos impresoras similares, se nos pide distruibuirlos de tal manera de conseguir la combinación de trabajos que generen el menor costo o presupuesto. Un enfoque goloso podría no encontrar solución porque no se puede conseguir

subestructura óptima con sólo analizar un subproblema, se deben analizar todas las combinaciones. Un algoritmo basado en backtracking eventualmente conseguiría la solución pero con una complejidad temporal del orden de 2^n, todos los presupuestos posibles.

Podemos intentar un mejor enfoque si dejamos de lado algunas variables. Es necesario obtener tanto las colas de trabajos como el costo total. Como las colas son intercambiables por las impresoras, alcanza con saber que son dos, sin mirar cual es cual. Para disminuir las combinaciones podríamos mirar, en cada nivel, el costo de imprimir el trabajo $t_{i,j}$ más alguna combinación mínima del nivel anterior. Llamamos $c(i, j)$ al costo de imprimir el trabajo T_j [1] luego del trabajo T_i, o sea, en la misma impresora. Se tiene que $j > i$ para todo trabajo, por la restricción de orden de impresión. Entonces $\{T_i, T_j\}$ son parte de una cola y $\{T_{i+1}, \ldots, T_{j-1}\}$ de la otra. Esta información extra nos permite dividir el «nivel anterior», o sea, los costos $c(k, j-1)$ (con $0 \leq k < j-2$), en dos subproblemas: si T_i y T_{i-1} van juntos o no. Cada subproblema genera una llamada recursiva, aunque no necesariamente el subproblema generado cuando T_i y T_{i-1} van en la misma cola generará la misma complejidad temporal que en el caso de que vayan en colas separadas.

Bajo este enfoque podemos definir $P(i, j)$ como el mínimo presupuesto que se genera al encolar los trabajos T_i y T_j, uno en cada impresora. Ello implica que los trabajos que ya estaban encolados lo fueron respetando este principio de optimalidad. El caso base se da al encolar el primer trabajo y es igual al costo $c(0, 1)$

$$P(1,0) = c(0,1) \qquad \{T_1\}, \quad \{\}$$

el cual es el único trabajo en una de las colas, mientras que la otra permanece vacía. Al encolar T_2 puede suceder que vaya junto a T_1 o no.

$$\begin{aligned} P(2,0) &= c(1,2) + c(0,1) && \{T_1, T_2\}, \quad \{\} \\ P(2,1) &= c(0,2) + c(0,1) && \{T_1\}, \quad \{T_2\} \end{aligned}$$

$P(2, 0)$ indica que en una cola se encoló último a T_2 mientras que la otra cola permanece vacía, y $P(2, 1)$, por el contrario, que se encoló uno en cada cola.

Al encolar el tercer trabajo va a pasar que la otra cola (en la que no se encola T_3) esté vacía o contenga como último trabajo a T_1 o a T_2. Si la «otra» cola está vacía, entonces en la de T_3 están T_1 y T_2. Si, en cambio, está T_1, en la cola de T_3 antes estaba T_2. En ambos casos existe sólo una posibilidad. Pero si en la cola que **no está** T_3 el último es T_2, entonces T_1 puede estar junto a T_2 o junto a T_3. Al tener dos posibilidades debemos quedarnos con la mejor, o sea, con aquella que genere el mínimo costo. La minimización va a venir dada por el costo del trabajo actual en la posición correspondiente sumado al presupuesto que debe elejirse al considerar esa posición. Esto resulta de la siguiente manera:

[1] $T_k = \text{VECTOR}(c(0, k), \ldots, c(k-1, k))$

$$P(3,0) = c(2,3) + c(1,2) + c(0,1)$$
$$P(3,0) = c(2,3) + P(2,0)$$
$$I_1 = \{T_1, T_2, T_3\}, I_2 = \{\}$$

$$P(3,1) = c(2,3) + c(0,2) + c(0,1)$$
$$P(3,1) = c(2,3) + P(2,1)$$
$$I_1 = \{T_1\}, I_2 = \{T_2, T_3\}$$

$$P(3,2) = \text{Mín}\,(c(0,3) + c(1,2) + c(0,1), c(1,3) + c(0,2) + c(0,1))$$
$$P(3,2) = \text{Mín}\,(c(0,3) + P(2,0), c(1,3) + P(2,1))$$
$$P(3,2) = \underset{k=0,1}{\text{Mín}}\,(c(k,3) + P(2,k))$$
$$I_1 = \{T_1, T_2\}, I_2 = \{T_3\} \vee I_1 = \{T_1, T_3\}, I_2 = \{T_2\}$$

Entonces, sucede que con un solo trabajo T_1 existe un único presupuesto. Con dos trabajo se dan dos combinaciones posibles de presupuesto, imprimir cada uno en una impresora distinta o los dos en la misma. Con tres trabajos se generan, en principio, las combinaciones resultantes con $P(3,0)$ y $P(3,1)$, para los cuales, en cada uno de ellos, existe sólo una combinación posible. La combinación dada por $P(3,2)$ resulta de elegir la mejor opción de las dos posibles. Esto hecho viene dado porque en $P(3,2)$ se produce un cambio en el encolamiento de impresoras. El trabajo T_3 no se imprime en la misma impresora que T_2, sino en la otra. En cambio, en los otros casos ($P(3,0)$ y $P(3,1)$) los trabajos T_2 y T_3 sí se imprimen juntos, entonces hay que mirar el caso en donde T_2 fue el último encolado.

En general se va a dar que cuando para $P(i,j)$ pase que $i-1 > j$, entonces va a haber una sola elección posible, según que haya pasado en la elección del nivel j. Pero cuando $i-1 = j$, va a ser necesario analizar todas las combinaciones del costo actual sumado al presupuesto del nivel anterior y entonces de todas ellas, quedarse con la menor. Generalizando la idea anterior, definimos una función recursiva de la forma:

$$P(i,j) = \begin{cases} c(0,1) & \text{si } i = 1, j = 0 \\ P(i-1,j) + c(i-1,i) & \text{si } i > 1, j < i-1 \\ \underset{0 \le k < j}{\text{Mín}}\,(P(j,k) + c(k,i)) & \text{si } i > 1, j = i-1 \end{cases} \tag{1}$$

la cual obtiene el mínimo presupuesto mirando desde el último trabajo encolado en cada impresora.

El caso base resulta trivial. El caso $j < i-1$ se da cuando la distancia entre el número de trabajo de una cola y el número de trabajo de la otra es mayor a 1. Esto significa que el trabajo T_i se está encolando justo después, y en la misma cola que T_{i-1}. Entonces el costo de T_i va a ser indefectiblemente $c(i-1,i)$ pues se imprime luego de T_{i-1}. El presupuesto acumulado será este costo más el presupuesto resultante de T_{i-1}, o sea, $P(i-1,j)$. Por el contrario, en el caso $j = i-1$ los trabajos T_i y T_j son cada uno el último de su cola. Es necesario poder conocer cual de los presupuestos que tienen como último en la cola a T_j resultará mínimo. Es necesario conocer todas las posibilidades del nivel anterior para poder tomar una decisión.

Observación. Cabe aclarar que la función $P(i,j)$ obtiene el mínimo presupuesto en base a los trabajos T_i y T_j, pero no es una solución al problema. La solución será dada luego de calcular todas las posibilidades de presupuestos al encolar el último trabajo, y luego de ello obtener el mínimo.

Cálculo y almacenamiento de presupuestos

La cantidad de costos posibles de cada trabajo viene dado por el número del mismo. Para el trabajo 1 existirá solo un costo, para el trabajo 2 los costos $c(0,2)$ y $c(1,2)$, para el trabajo n habrá n costos, $c(0,n),\ldots,c(n-1,n)$. Dado el uso de índices, almacenar los datos en una matriz resulta un buen acercamiento, sobre todo para el hecho que consultar el dato en este tipo de estructuras resulta constante y el mapeo sale directo. Como los índices se mueven entre $i=0,\ldots,n-1$ y $j=1,\ldots,n$, resulta práctico utilizar una matriz de $(n+1)\times(n+1)$ aunque «sobren» una fila y una columna. De esta manera se obtiene una matriz triangular superior con ceros en y por debajo de la diagonal, de la forma

$$C = \begin{pmatrix} 0 & c_{0,1} & c_{0,2} & \cdots & c_{0,n-1} & c_{0,n} \\ 0 & 0 & c_{1,1} & \cdots & c_{1,n-1} & c_{1,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & c_{n-1,n-1} & c_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}^{(n+1)\times(n+1)}$$

Como los presupuestos $P(i,j)$ también varían entre $1,\ldots,n$ y $0,\ldots,n-1$ respectivamente, y además los índices están «invertidos» con respecto a $c(i,j)$ [2]. Este hecho permite almacenar los valores de $P(i,j)$ [3] en la parte triangular inferior de la matriz C [4], de la forma

$$C = \begin{pmatrix} 0 & c_{0,1} & c_{0,2} & \cdots & c_{0,n-2} & c_{0,n-1} & c_{0,n} \\ P_{1,0} & 0 & c_{1,2} & \cdots & c_{1,n-2} & c_{1,n-1} & c_{1,n} \\ P_{2,0} & P_{2,1} & 0 & \cdots & c_{2,n-2} & c_{2,n-1} & c_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ P_{n-2,0} & P_{n-2,1} & P_{n-2,2} & \cdots & 0 & c_{n-2,n-1} & c_{n-2,n} \\ P_{n-1,0} & P_{n-1,1} & P_{n-1,2} & \cdots & P_{n-1,n-2} & 0 & c_{n-1,n} \\ P_{n,0} & P_{n,1} & P_{n,2} & \cdots & P_{n,n-2} & P_{n,n-1} & 0 \end{pmatrix}^{(n+1)\times(n+1)}$$

Obteniendo el mínimo entre $P(n,0),\ldots P(n,n-1)$ se consigue el mejor presupuesto final. Dado que para ello se van a tener que calcular todos los $P(i,j)$, es indistinto desde que lugar se comience. De todas formas, una estrategia práctica podría ser comenzar desde $P(n,n-1)$, quien, al quedarse con el mínimo $P(n-1,k)+c(k,n)$ tal que $0\leq k<n-1$, va a calcular recursivamente todos los presupuestos pertinentes al trabajo anterior. Si aplicamos la misma estrategia, o sea, seguir la recursión por $P(q,q-1)$ en cada trabajo T_q, al momento de solicitar el presupuesto $P(q,r)$ para todo $r:0\leq r<q-1$, ya habrán sido calculados todos los presupuestos correspondientes a las combinaciones generadas hasta el trabajo T_{q-1}.

[2] se hace un abuso de notación entre $c(i,j)$ y $c_{i,j}$ según el contexto.

[3] se hace un abuso de notación entre $P(i,j)$ y $P_{i,j}$ según el contexto.

[4] si bien se puede utilizar otra matriz para almacenar los presupuestos, el hecho de utilizar los índices «invertidos» permite almacenar tanto los costos como los presupuestos en una misma matriz, con el fin de optimizar recursos.

Esta estrategia de generar el universo de soluciones pero guardando información para no tener que volver a calcular los mismos datos una y otra vez se la conoce como *Programación Dinámica*[1, Cap.8, Dynamic Programming].

Ejemplo. Dada una entrada tal que

$$\begin{aligned} n &= 4 \\ T_1 &= (9) \\ T_2 &= (3, 8) \\ T_3 &= (2, 1, 4) \\ T_4 &= (7, 5, 7, 8) \end{aligned}$$

resulta

$$C = \begin{pmatrix} 0 & c_{0,1} & c_{0,2} & c_{0,3} & c_{0,4} \\ P_{1,0} & 0 & c_{1,2} & c_{1,3} & c_{1,4} \\ P_{2,0} & P_{2,1} & 0 & c_{2,3} & c_{2,4} \\ P_{3,0} & P_{3,1} & P_{3,2} & 0 & c_{3,4} \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & 0 \end{pmatrix} = \begin{pmatrix} 0 & 9 & 3 & 2 & 7 \\ P_{1,0} & 0 & 8 & 1 & 5 \\ P_{2,0} & P_{2,1} & 0 & 4 & 7 \\ P_{3,0} & P_{3,1} & P_{3,2} & 0 & 8 \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & 0 \end{pmatrix}$$

El problema se resuelve al obtener el mínimo entre $P(4,0) \dots P(4,3)$. Para ello será necesario calcular no sólo la fila 4 sino todos los valores de P. Respetando lo descripto de ir recorriendo por la «diagonal», comenzamos por $P(4,3)$

$$\begin{aligned} P(4,3) &= \text{Mín}\left(P(3,0) + c(0,4), P(3,1) + c(1,4), P(3,2) + c(2,4)\right) \\ P(4,3) &= \text{Mín}\left(P(3,0) + 7, P(3,1) + 5, P(3,2) + 7\right) \end{aligned}$$

con lo cual es necesario calcular $P(3,0)$, $P(3,1)$ y $P(3,2)$. El próximo a calcular es $P(3,2)$

$$\begin{aligned} P(3,2) &= \text{Mín}\left(P(2,0) + c(0,3), P(2,1) + c(1,3)\right) \\ P(3,2) &= \text{Mín}\left(P(2,0) + 2, P(2,1) + 1\right) \end{aligned}$$

necesitando los cálculos correspondientes a $P(2,0)$ y $P(2,1)$

$$\begin{aligned} P(2,1) &= \text{Mín}\left(P(1,0) + c(0,2)\right) \\ P(2,1) &= P(1,0) + 3 \\ P(1,0) &= c(0,1) = 9 \\ P(2,1) &= 9 + 3 = 12 \end{aligned}$$

Al calcular $P(2,0)$ no es necesario volver a calcular $P(1,0)$

$$P(2,0) = \text{Mín}\left(P(1,0) + c(1,2)\right) = 9 + 8 = 17$$

$$C = \begin{pmatrix} 0 & 9 & 3 & 2 & 7 \\ 9 & 0 & 8 & 1 & 5 \\ 17 & 12 & 0 & 4 & 7 \\ P_{3,0} & P_{3,1} & P_{3,2} & 0 & 8 \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & 0 \end{pmatrix}$$

Con lo ya calculado, obtenemos $P(3,2)$. Para $P(4,3)$ aún es necesario calcular $P(3,1)$ y $P(3,0)$, los cuales se pueden obtener de forma directa ya que sus niveles anterior ya están calculados

$$P(3,2) = \text{Mín}\,(17+2, 12+1) = 13$$
$$P(3,1) = P(2,1) + c(2,3) = 12+4 = 16$$
$$P(3,0) = P(2,0) + c(2,3) = 17+4 = 21$$

$$C = \begin{pmatrix} 0 & 9 & 3 & 2 & 7 \\ 9 & 0 & 8 & 1 & 5 \\ 17 & 12 & 0 & 4 & 7 \\ 21 & 16 & 13 & 0 & 8 \\ P_{4,0} & P_{4,1} & P_{4,2} & P_{4,3} & 0 \end{pmatrix}$$

Ahora sí podemos obtener $P(4,3)$

$$P(4,3) = \text{Mín}\,(P(3,0)+7, P(3,1)+5, P(3,2)+7)$$
$$P(4,3) = \text{Mín}\,(21+7, 16+5, 13+7) = 20$$

$$C = \begin{pmatrix} 0 & 9 & 3 & 2 & 7 \\ 9 & 0 & 8 & 1 & 5 \\ 17 & 12 & 0 & 4 & 7 \\ 21 & 16 & 13 & 0 & 8 \\ P_{4,0} & P_{4,1} & P_{4,2} & 20 & 0 \end{pmatrix}$$

Para resolver el problema aún es necesario calcular $P(4,2)$, $P(4,1)$ y $P(4,0)$, pero todos los presupuestos que necesitan ya están calculados, con lo cual

$$P(4,2) = P(3,2) + c(3,4) = 13+8 = 21$$
$$P(4,1) = P(3,1) + c(3,4) = 16+8 = 22$$
$$P(4,0) = P(3,0) + c(3,4) = 21+8 = 29$$

$$C = \begin{pmatrix} 0 & 9 & 3 & 2 & 7 \\ 9 & 0 & 8 & 1 & 5 \\ 17 & 12 & 0 & 4 & 7 \\ 21 & 16 & 13 & 0 & 8 \\ 29 & 22 & 21 & 20 & 0 \end{pmatrix}$$

Siendo el mínimo presupuesto igual a 20, el correspondiente a imprimir como último en una cola el trabajo T_4 y como último en la otra cola el trabajo T_3. Para terminar de determinar las secuencias de trabajos impresos es necesario volver recorrer los presupuestos, pero esta

vez desplegando hasta los costos, o sea

$$\begin{aligned}
\text{Mín}\,(P(4,k)) &= P(4,3) \\
&= \text{Mín}\,(P(3,k) + c(k,4)) \\
&= P(3,2) + c(2,4) \\
&= [\text{Mín}\,(P(2,k) + c(k,3))] + c(2,4) \\
&= P(2,1) + c(1,3) + c(2,4) \\
&= [\text{Mín}\,(P(1,k) + c(k,2))] + c(1,3) + c(2,4) \\
&= P(1,0) + c(0,2) + c(1,3) + c(2,4) \\
&= c(0,1) + c(0,2) + c(1,3) + c(2,4)
\end{aligned}$$

con lo cual, las colas de los trabajos que generan el mejor presupuesto quedan

$$I_1 = \{T_1, T_3\}, I_2 = \{T_2, T_4\}$$

Se puede ver que una vez obtenido el mejor presupuesto se pueden recorrer recursivamente los resultados, guardando en el camino los costos de aquellos trabajos que fueron los que generaron el mínimo presupuesto. Dado que se pide devolver los indices de una de las colas de impresión, un procedimiento posible puede ser, a medida que se desarrolla la recursión, ir encolando los pares de índices que representan el costo de imprimir el trabajo j luego del trabajo i. En el ejemplo quedaría

$$\{(0,1),(0,2),(1,3),(2,4)\}$$

Luego, para quedarse con una de las colas, el procedimiento sería el siguiente: Defino Q como una de las colas, luego tomo $(0,1)$ y encolo 1 en Q. Tomo $(0,2)$, como $0 \neq 1$, lo descarto y tomo $(1,3)$, como $1 = 1$ encolo 3. Tomo $(2,4)$, como $2 \neq 3$, descarto. Llegué al final, terminé y obtuve

$$Q = \{1,3\}$$

Alcanza con devolver la cola Q pues quienes no estén en Q estarán en Q', pues

$$Q' = \{1,\ldots,n\} \setminus Q$$

Hipótesis de solución

El presupuesto $P(i,j)$ en la *Función* (1) obtiene el menor presupuesto para algún par de trabajos T_i, T_j.

Para resolver el problema planteado es necesario obtener todos los menores presupuestos para el último trabajo y luego, de todos ellos, quedarse con aquel que sea el menor de todos.

Teorema 2.2.0.1. *Dados $T_1,\ldots,T_n$ trabajos, el menor presupuesto generado a partir de los costos de impresión de cada trabajo está dado por*

$$\underset{0\leq i<n}{\text{Mín}}\,(P(n,i))$$

En el *Algoritmo 1* se desarrolla la implementación del procedimiento descripto.

Algoritmo 1 Presupuesto Óptimo

Entrada:
$n \leftarrow$ Cantidad de Trabajos
$Trabajos \leftarrow$ Vector Trabajos$(T_1, \ldots, T_n)$ ▷ $T_k = (c_{0,k}, \ldots, c_{k-1,k})$, *costos*

Salida:
Costo Total ▷ C
Una Cola de Trabajos ▷ $L = i_1 \ldots i_k$
Cantidad Encolados ▷ k

1: $A \leftarrow$ Cargar Matriz$(A, Trabajos, n)$ ▷ $\mathcal{O}(n^2)$
2: $C \leftarrow$ Mínimo Presupuesto(A, n) ▷ $\mathcal{O}(n^2)$
3: $L \leftarrow$ Cola de Trabajos(A, n) ▷ $\mathcal{O}(n^2)$
4: $k \leftarrow$ Cantidad(L) ▷ $\mathcal{O}(1)$
5: **retornar** C, k, L ▷ *final algoritmo* $\mathcal{O}(n^2)$

6: **función** Cargar Matriz(A, T, n)
7: **para** i **desde** 1 **hasta** n **hacer** ▷ $\mathcal{O}(n)$
8: **para** j **desde** 0 **hasta** $i-1$ **hacer** ▷ $\mathcal{O}(n)$
9: $A_{j,i} \leftarrow T_{j,i}$ ▷ *cargar el costo* $c_{j,i}$
10: $A_{i,j} \leftarrow NULO$ ▷ *los presupuestos inician nulos*
11: **retornar** A
12: **fin función** ▷ *final función* $\mathcal{O}(n^2)$

13: **función** Mínimo Presupuesto(A, n)
14: $minimo \leftarrow \infty$
15: **para** j **desde** $n-1$ **hasta** 0 **hacer** ▷ $\mathcal{O}(n)$
16: $A_{n,j} \leftarrow$ Mejor Presupuesto(A, n, j) ▷ $\mathcal{O}(n^2)$ *si* $j = n-1$ *sino* $\mathcal{O}(1)$
17: **si** $A_{n,j} < minimo$ **entonces**
18: $minimo \leftarrow A_{n,j}$
19: **retornar** $minimo$
20: **fin función** ▷ *ciclo* $\mathcal{O}(1 * n^2 + (n-1) * 1)$, *final función* $\mathcal{O}(n^2)$

21: **función** Mejor Presupuesto(A, i, j) ▷ *función* $P(i, j)$
22: **si** $i, j \leq 0 \vee i \leq j$ **entonces**
23: **retornar** ∞
24: **si** $i = 1$ **entonces**
25: **retornar** $A_{0,1}$ ▷ $\mathcal{O}(1)$
26: **si** $j < i-1$ **entonces**
27: **si** $A_{i-1,j}$ **es** $NULO$ **entonces**
28: $A_{i-1,j} \leftarrow$ Mejor Presupuesto$(A, i-1, j)$
29: **retornar** $A_{i-1,j} + A_{i-1,i}$
30: **si** $j = i-1$ **entonces**
31: $min \leftarrow \infty$
32: **para** k **desde** $i-2$ **hasta** 0 **hacer** ▷ $\mathcal{O}(i)$
33: **si** $A_{i-1,k}$ **es** $NULO$ **entonces**
34: $A_{i-1,k} \leftarrow$ Mejor Presupuesto$(A, i-1, k)$
35: **si** $A_{i-1,k} + A_{k,i} < min$ **entonces**
36: $min \leftarrow A_{i-1,k} + A_{k,i}$
37: **retornar** min
38: **fin función** ▷ *final función* $\mathcal{O}(i^2) \vee \mathcal{O}(i) \vee \mathcal{O}(1)$

Algoritmo 2 Presupuesto Óptimo (continuación)

```
39: función Cola de Trabajos(A, n)
40:     min ← ∞
41:     iMin ← ∞
42:     para j desde 0 hasta n − 1 hacer                          ▷ ciclo O(n)
43:         p ← Presupuesto(A, n, j)                              ▷ ya calculado, O(1)
44:         si p < min entonces
45:             min ← p
46:             iMin ← j
47:     lista ← Lista de Indices(A, lista, n, iMin)               ▷ O(n²)
48:     Q ← Cola
49:     iPrev ← 0
50:     para cada elem ← Proximo(lista) hacer                     ▷ ciclo O(n)
51:         si elem[0] = iPrev entonces
52:             Q • elem[1]
53:             iPrev = elem[1]
54:     retornar Q
55: fin función                                                   ▷ final función O(n²)

56: función Lista de Indices(A, lista, i, j)
57:     si i = 1 ∧ j = 0 entonces
58:         retornar (0, 1) • lista
59:     si j < i − 1 entonces
60:         retornar Lista de Indices(A, (i − 1, i) • lista, i − 1, j)
61:     si j = i − 1 entonces
62:         iMin ← ∞
63:         min ← ∞
64:         para k desde i − 2 hasta 0 hacer                      ▷ ciclo O(i)
65:             si A_{i−1,k} + A_{k,i} < min entonces             ▷ calculada, O(1)
66:                 iMin ← k
67:                 min ← A_{i−1,k} + A_{k,i}
68:         retornar Lista de Indices(A, (iMin, i) • lista, j, iMin)
69:     retornar lista
70: fin función                                                   ▷ final función O(i²)
```

2.3 Justificación formal de correctitud

Queremos probar que *Algoritmo 1* resuelve el problema, o sea que, dados n trabajos se obtiene el menor presupuesto, el cual será aquel que minimice las combinaciones de costos de impresión de cada trabajo según las restricciones impuestas, como ya ha sido descripto.

Planteamos que el algoritmo es una implementación correcta del teorema 2.2.0.1. Si no lo fuese, no resuelve el problema. Con lo cual, vamos a querer demostrar que vale el teorema.

Demostración. Sean $P(n,0),\ldots,P(n,n-1)$ los menores presupuestos tal que se cubren todas las posibilidades de presupuestos en donde se imprime como último trabajo al trabajo T_n en una de las colas.

$$\begin{aligned} &I_1 = \{T_1,\ldots,T_n\}, \quad I_2 = \{\} \\ &\vdots \\ &I_1 = \{\ldots,T_n\}, \quad I_2 = \{\ldots,T_{n-1}\} \end{aligned}$$

Luego se toma el más chico y ése es el presupuesto mínimo para n trabajos. □

Lema 2.3.0.2. $(\forall i,j \ : i>j, j\geq 0)$ $P(i,j)$ *obtiene el menor presupuesto de los costos de impresión de los trabajos* $T_1,\ldots,T_i$ *tal que* T_i *es el último encolado en una de las colas y* T_j *el último encolado en la otra cola.*

$$P(i,j) = \begin{cases} c(0,1) & \text{si } i=1, j=0 \\ P(i-1,j)+c(i-1,i) & \text{si } i>1, j<i-1 \\ \underset{0\leq k<j}{\text{Mín}}\,(P(j,k)+c(k,i)) & \text{si } i>1, j=i-1 \end{cases}$$

Demostración.

Caso 1: $i=1, j=0$. Es el costo de imprimir el único trabajo T_1 en una de las colas, mientras que la otra queda vacía. Se ve que $c(0,1)$ es trivialmente mínimo.

Caso 2: $i>1, j<i-1$. Como T_i es el último de una cola y T_j el último de la otra, y además entre i y j hay una distancia mayor a 1, lo que sucede es que T_i se imprime en la misma cola que $T_{i-1},\ldots,T_{j+1}$. O sea, los últimos $i-j$ trabajos están encolados juntos.

$$I_1 = \{\ldots,T_4,T_5,T_6\}, \quad I_2 = \{\ldots,T_3\}$$

Entonces se imprime T_i justo después de T_{i-1}, con lo cual, el costo de impresión de T_i es $c(i-1,i)$. En este caso el mínimo presupuesto va a estar dado por ese costo *más* el mínimo presupuesto que se genera al imprimir como último a T_{i-1} en una cola y a T_j en la otra.

$$I_1 = \{\ldots,T_4,T_5\}, \quad I_2 = \{\ldots,T_3\}$$

O sea, $P(i-1,j)+c(i-1,i)$.

Caso 3: $i>1, j=i-1$. En este caso T_i se imprime en una cola y T_{i-1} en la otra.

$$I_1 = \{\ldots,T_6\}, \quad I_2 = \{\ldots,T_5\}$$

No se conoce el costo de impresión de T_i porque no se sabe luego de que trabajo hay que imprimirlo. Como puede ser cualquiera de los menores que T_{i-1}, hay que analizarlos todos para poder obtener el mejor. Dado que T_i se puede imprimir luego de cualquier $T_{i-2}, \ldots, T_1$, obtener el mínimo presupuesto consiste en minimizar la *suma* del costo de imprimir T_i luego de algún T_k con el mínimo presupuesto que se genera al imprimir como último a T_k en una cola y a T_{i-1} en la otra.

$$P(5,4) = \text{Mín} \begin{cases} P(4,3) + c(3,5) \\ P(4,2) + c(2,5) \\ P(4,1) + c(1,5) \\ P(4,0) + c(0,5) \end{cases}$$

O sea, $\underset{0 \leq k < j}{\text{Mín}} (P(j,k) + c(k,i))$.
Como cada presupuesto anterior se calcula recursivamente, la minimización vale desde el punto que a cada paso se obtiene el mínimo posible, siendo eventualmente el caso base. □

2.4 Cota de complejidad temporal

La idea general del *Algoritmo 1* se basa en la ventaja que se obtiene al guardar determinados valores que al momento de calcularlos toman una complejidad grande, de menera que si los necesitamos varias veces, no es necesario volver a calcularlos, disminuyendo la cota de complejidad temporal.

Dada una entrada de n trabajos se tienen $n * (n+1)/2$ costos de trabajos. La función Cargar Matriz es trivialmente cuadrática. Mínimo Presupuesto itera n veces sobre una función, Mejor Presupuesto, con lo cual el costo es $\mathcal{O}(n) * \mathcal{O}(\text{Mejor Presupuesto})$. Como Mejor Presupuesto la primera vez es cuadrática y el resto de las veces es constante, entonces el ciclo tiene una entrada cuadrática y $n-1$ entradas constantes. Tal como se muestra en el pseudocódigo, $(1 * n + (n-1) * 1) \in \mathcal{O}(n^2)$.

Hay que ver que efectivamente Mejor Presupuesto es cuadrática. Esta función es recursiva, guardando lo ya calculado en una matriz auxiliar de manera de no tener que realizar las mismas operaciones más de una vez. El caso base se da cuando los índices pasados son $(1,0)$, devolviendo el valor almacenado en la matriz de forma constante.

En los otros dos casos se piden valores de la matriz que pueden estar calculados o no. Si lo están, es constante. Si no lo están, la llamada recursiva definirá el costo. Como existen muchas llamadas que usan los mismos valores, la primera que lo llame va a pagar el costo y las demás no. Los valores a calcular son n^2, de manera que de las n^2 que se va a llamar a Mejor Presupuesto (pues es necesario analizar todos los presupuestos), algunas tendrán costo constante. La forma de recorrer la matriz en busca de los valores tiene la ventaja de permitir observar con mayor facilidad la distintición entre las llamadas constantes y aquellas con algún costo.

Cuando se llamada a la función desde $n, n-1$, la matriz de valores está vacía. Este pedido necesita todos los valores de la fila $n-1$. Para ello comienza a recorrer desde $n-2$, con lo cual vuelve a suceder lo mismo. El procedimiento continúa buscando hasta llegar al caso base. A partir de ahí, los retornos de las llamadas recursivas van completando la matriz hasta el lugar del pedido original, $n, n-1$. Los valores de $(n, n-1), \ldots, (n, 0)$ continúan vacíos, pero el resto de la matriz está completa. Se ve entonces que el llamado a Mejor Presupuesto$(n, n-1)$

tiene complejidad $\Theta(n^2)$ puedo recorre toda la matriz. El resto de los $n-1$ llamados sólo tienen que obtener el valor calculado, con lo cual tienen costo constante.

La función Cola de Trabajos tiene dos partes. Primero va recorriendo la matriz y quedandose con aquellos pares de índices que representan los trabajos que se imprimen luego de otros tal que son aquellos que generan el presupuesto óptimo. Esto lo hace a través de la función auxiliar Lista de Índices, la cual recorre toda la matriz con costo cuadrático. La lista resultante tiene n pares de índices, uno por cada costo de cada uno de los trabajos. Después de obtener la lista, se la recorre para quedarse con cada índice de trabajo que se imprime en una de las impresoras. El costo de recorrer la lista es lineal. Con lo cual, esta función tiene costo $\mathcal{O}(n^2)$. Obtener la cantidad de elementos de una lista es constante por implementación [5].

2.5 Verificación mediante casos de prueba

Los siguientes casos de prueba fueron desarrollados para intentar cubrir la mayor cantidad de posibilidades para cantidades de trabajos chicas, de manera de poder analizar el correcto comportamiento del algoritmo.

Prueba 1

```
Entrada:          Elección:          Salida:
1                                    10 1 1
10                (10)
```

Prueba 2

```
Entrada:          Elección:          Salida:
2                                    30 1 1
10                (10)
20 30             (20) 30
```

Prueba 3

```
Entrada:          Elección:          Salida:
2                                    30 2 1 2
10                (10)
30 20              30 (20)
```

[5] se utiliza `std::vector` [3]

Prueba 4

```
Entrada:          Elección:          Salida:
3                                    90 2 1 2
10                (10)
20 30              20 (30)
50 80 90          (50) 80 90
```

Prueba 5

```
Entrada:          Elección:          Salida:
4                                    135 2 1 2
10                (10)
10 50              10 (50)
10 80 90          (10) 80 90
10 70 75 65        10  70 75 (65)
```

Prueba 6

```
Entrada:          Elección:          Salida:
4                                    180 2 1 3
10                (10)
20 50             (20) 50
30 60 80           30 (60) 80
40 70 90 100       40  70 (90) 100
```

Prueba 7

```
Entrada:          Elección:          Salida:
4                                    45 3 1 2 3
10                (10)
90 10              90 (10)
90 90 10           90  90 (10)
15 15 15 100      (15) 15  15 100
```

Prueba 8

```
Entrada:          Elección:          Salida:
5                                    5 5 1 2 3 4 5
1                 (1)
```

```
1 1                 1 (1)
1 1 1               1  1 (1)
1 1 1 1             1  1  1 (1)
1 1 1 1 1           1  1  1  1 (1)
```

2.6 Medición empírica de la performance

El siguiente gráfico muestra el análisis de complejidad del algoritmo midiendo tanto su tiempo de ejecución como la cantidad de operaciones realizadas, las cuales fueron medidas en base a la cantidad de llamadas recursivas a la funciones Mejor Presupuesto y Cola de Trabajos del algoritmo 1.

Se generó 1 archivo de entrada para cada $n \in \{100, 120, 140, \ldots, 1460, 1480, 1500\}$ cantidad de trabajos. Dentro de cada archivo se generaron 500 inputs de entrada, todos con igual cantidad de trabajos n. Los costos de impresión de cada trabajo se generaron de manera pseudo-aleatoria utilizando una distribución uniforme en el rango $[1 \ldots 500]$.

Se midieron, para cada entrada, los tiempos de ejecución del algoritmo y la cantidad de llamadas recursivas, y luego se obtuvieron los promedios (para cada n).

La medición de cantidad de llamadas recursivas sirve para realizar un acercamiento empírico-teórico al orden de complejidad temporal. Admitiendo que el tiempo de ejecución de cada llamada es idealmente invariable, podemos establecer que el tiempo total de ejecución del algoritmo será proporcional a esta medición, y que los «picos» (variaciones en la performance) observados en la medición empírica de tiempos de ejecución sean probablemente causados por cuestiones arbitrarias en gran medida ajenas al algoritmo, o potencialmente inherentes pero imprevisibles/inevitables (i.e., interrupciones del scheduler del sistema operativo, swapeo de memoria, hits/misses en la caché del procesador, etc). Se puede determinar, por lo tanto, que ambas mediciones se encuentran acotadas superior e inferiormente por la complejidad asintótica $\mathcal{O}(n^2)$.

No se midieron peores o mejores casos dado que, por el propio comportamiento del algoritmo, el mismo siempre se ejecuta regido por una complejidad cuadrática, tanto para conseguir el presupuesto como para devolver la cola.

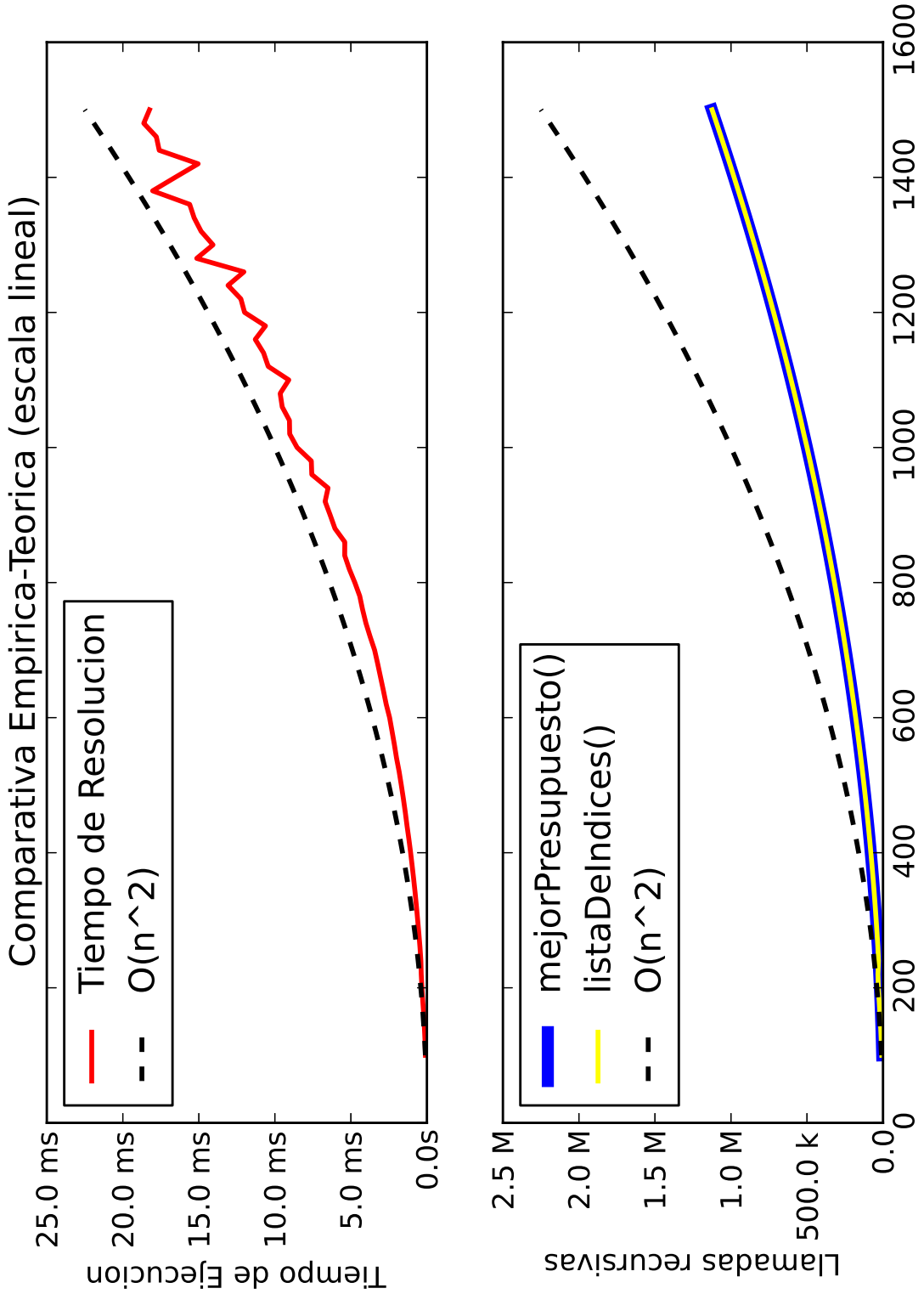

Figura 3: **Mediciones: tiempo de ejecución y cantidad de llamadas recursivas**

3 Problema 2: Replicación de contenido

3.1 Descripción

Presentación: Red de servidores - CDN

Cierta organización hace uso de una *Red de Entrega de Contenidos en Internet* mediante la cual se logra proveer la información eficientemente, utilizando para ello una red de servidores interconectados y localizados en distintos puntos estratégicos del planeta, logrando así que el envío de datos sea siempre efectuado desde el servidor geográficamente más cercano a cada cliente, y asegurando de esta forma minimizar la latencia producida por el *tiempo de propagación*[6].

La empresa cuenta actualmente con n servidores conectados a través de m enlaces bidireccionales de alta velocidad, de manera tal que buscan asegurarse de que cada servidor dispone en todo momento de una réplica del mismo contenido con el que cuentan los demás. La utilización de cada uno de estos enlaces tiene un costo determinado, el cual no necesariamente es el mismo, para cada uno de ellos.

Objetivo: Sistema de Replicación

La compañía busca implementar un *Sistema de Replicación* mediante el cual puedan distribuir los datos, ya no entre sus distintos clientes, sino dentro de su propia red de servidores. Para ello, utilizarán la metodología de *Replicación Maestro-Esclavo*, la cual requiere que alguno de los servidores cumpla el rol de *master*, de manera que ante la presencia de contenidos modificados este sea el encargado de comandar la sincronización de los datos hacia el resto.

Para el buen funcionamiento del sistema, es necesario que sean habilitados tantos enlaces como para asegurar que desde cualquiera de los servidores exista al menos una ruta hacia el *master*. Por las características propias de la tecnología de fibra óptica utilizada en los *enlaces backbone*, el retardo generado por la distancia física que debe recorrer uno de estos resulta despreciable frente al tiempo total de transmisión de los datos, el cual resulta a su vez afectado por otras variables. En consecuencia y a efectos prácticos, se considera que el tiempo que demora la información en desplazarse entre los servidores que conecta un determinado enlace es constante para cualquiera de ellos.

En la Figura 4 se pueden observar 2 configuraciones de servidores y enlaces. La primer configuración forma un grafo conexo, es decir no tenemos servidores aislados, por lo que el problema esta bien definido. En la segunda configuración se puede observar que el servidor *S1* esta aislado, el problema no está bien definido dado que no existe una solución que cumpla con lo pedido.

[6] http://es.wikipedia.org/wiki/Tiempo_de_propagacion

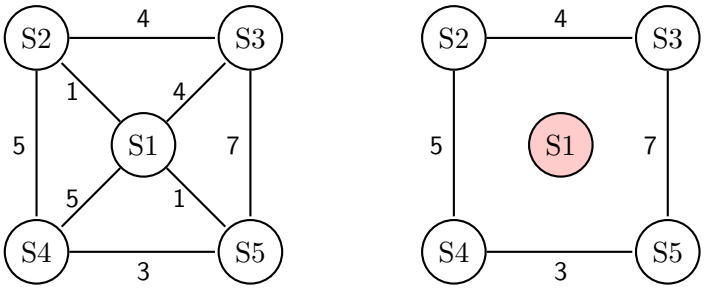

Figura 4: **Dos configuraciones distintivas de servidores y enlaces**

Por diversas razones se decidió tercerizar el trabajo de cálculo que les permitirá saber el modo y la forma en que deberán utilizar las conexiones para lograr que la implementación funcione de forma eficiente. Para esto, han dividido la tarea en dos soluciones complementarias, encomendando cada una de ellas a consultoras independientes.

- A la primera de ellas, le han solicitado disminuir al mínimo posible la cantidad de enlaces entre servidores de manera tal de mantener la conectividad necesaria para el funcionamiento del sistema, logrando así la máxima reducción posible en el gasto ocasionado por la utilización de los mismos.
- A la segunda consultora, le fue encargada la tarea de idear una técnica para seleccionar el servidor que cumplirá el papel de *master*, de manera tal que el tiempo necesario para una replicación completa sea mínimo.

Si bien los costos de cada uno de los enlaces es distinto, la velocidad de transmisión de los datos es igual para todos. De esta manera, tiempo mínimo hace referencia a menor cantidad de pasos desde el servidor principal hasta aquel más lejano. Los enlaces son bidireccionales, con lo cual no hay restricción de orden entre los servidores. Se puede iniciar desde cualquiera, siempre y cuando se pueda llegar a todos. Si un servidor está enlazado a más de un servidor, la replicación es simultánea a todos sus vecinos. En la figura 5 se puede observar la solución obtenidas en 2 etapas para la primera configuración de la figura 4.

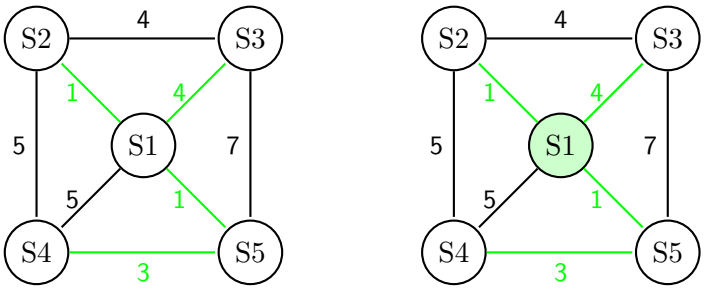

Figura 5: **Configuración de servidores. En verde los enlaces que minimizan el costo y el servidor seleccionado como master**

Para algunas configuraciones de servidores y enlaces es posible que existan múltiples soluciones óptimas, en ese caso solo se requiere una de las mismas.

Se requiere cumplir individualmente el papel de las dos consultoras, brindando los algoritmos necesarios, respetando una complejidad de $\mathcal{O}(n^3)$ para el primer caso y de $\mathcal{O}(n)$ para el otro, siendo n la cantidad de servidores.

3.2 Hipótesis de resolución

Consultora 1: selección de enlaces

Modelaremos el problema con un grafo $G = (V, E)$, donde cada nodo representa un servidor y cada arista un enlace entre 2 servidores.

Definición 3.2.0.1. Sea $\phi : E \to \mathbb{R}_{>0}$ una función tal que para cada arista $e \in E$ devuelve el costo de transmisión de dicho enlace.

Se necesita encontrar el conjunto de aristas E' tal que el grafo $T = (V, E')$ sea conexo y que para todo sub-grafo $G' = (V, E'')$ conexo se cumpla

$$\sum_{\forall e \in E'} \phi(e) \leq \sum_{\forall e \in E''} \phi(e)$$

Es decir, necesitamos un sub-grafo de G donde no queden servidores aislados y que el costo de conexión entre los mismos sea el mínimo posible. T tiene que ser un grafo sin ciclos porque de lo contrario sería posible minimizarlo quitando una de las aristas del ciclo. T es un árbol de G, con lo cual, al agregar cualquier arista $e \in E$ que no esté en E' va a generar un ciclo simple en T. Es mínimo porque para cualquier árbol generador de G, T es aquel de menor costo. Es a lo que se le llama *Árbol Generador Mínimo (AGM)* de un grafo G.

Una manera de obtener el AGM de un grafo es utilizando un algoritmo goloso que recorra el grafo y a cada paso vaya guardando aquel par de nodos tal que la arista que los conecta es la «menor». Lo que se elige como menor es lo que diferencia a los algoritmos de Kruskal y de Prim.

El algoritmo de Prim comienza por un nodo cualquiera y lo marca. Luego, de todas sus aristas adyacentes se queda con la de menor peso, marcando nodo con el que se conecta mediante esa arista. Repite el procedimiento sumando a cada paso las aristas adyacentes del nuevo nodo marcado. Termina cuando todos los nodos han sido marcados. Kruskal, por otro lado, en cada paso mira la totalidad de las aristas y se queda con la menor entre las que conectan un nodo marcado con uno no marcado. En cada paso Prim obtiene un árbol generador mínimo de un subgrafo de G. Kruskal, en cambios, va obteniendo bosques donde sus componentes conexas son árboles generadores mínimos de subgrafos de G.

Ambos algoritmos tienen el mismo orden de complejidad, elegimos implementar Kruskal porque utiliza estructuras menos complejas. En el algoritmo 3 [7] se puede ver la implementación.

[7] basado en Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2009, Third Edition, página 631

La complejidad del algoritmo de Kruskal puede variar dependiendo de las estructuras de datos y de los algoritmos utilizados para ordenar las aristas y a su vez mantener los conjuntos disjuntos que contienen a los sub-arboles. Para ordenar las aristas se utiliza un *heap* ordenado mediante el algoritmo *heapsort* de la biblioteca estándar de C++. Para implementar conjuntos disjuntos se implementan las heurísticas *union by rank* [8] y *path compression* [9], las cuales permiten lograr una complejidad del orden de $\mathcal{O}(m)$ sobre la cantidad de operaciones.

Algoritmo 3 AGM-Kruskal

Entrada:
G = (V,E)
Salida:
A: conjunto de aristas del AGM

▷ *Los siguiente arreglos se ultizan para implementar Conjuntos Disjuntos*
height[1...N]: Vector de alturas
set[1...N]: Vector de padres
1: $A = \emptyset$
2: **para cada** nodo v ∈ G.V **hacer**
3: height[v] ← 0
4: set[v] ← v
5: OrdenarPorPeso(G.E)
6: **para cada** arista (u,v) ∈ G.E,tomadas en orden creciente por peso **hacer**
7: **si** DevolverConjunto(u) <> DevolverConjunto(v) **entonces**
8: $A = A \cup \{(u, v)\}$
9: UnirConjuntos(u,v)
10: **función** DevolverConjunto(v)
11: r ← v
12: **mientras** set[r] ≠ r **hacer**
13: r ← set[r]
14: i ← v
15: **mientras** i ≠ r **hacer**
16: j ← set[i]
17: set[i] ← r
18: i ← j
19: **retornar** r
20: **fin función**
21: **función** UnirConjuntos(u,v)
22: **si** height[u] = height[v] **entonces**
23: height[u] ← height[u] + 1
24: set[v] ← u
25: **sino**
26: **si** height[u] > height[v] **entonces**
27: set[v] ← u
28: **sino**
29: set[u] ← v
30: **fin función**

[8] Brassard, Bratley, *Fundamental of Algorithmics*, Prentice Hall, 1996,página 178
[9] Brassard, Bratley, *Fundamental of Algorithmics*, Prentice Hall, 1996,página 179

Consultora 2: elección del maestro

Utilizando el árbol generador T recibido de la consultora 1 tenemos que encontrar aquel nodo considerado como «maestro».

Definición 3.2.0.2. Sea $n \in V$ un nodo de T, n es *maestro* si para todo árbol T_i isomorfo con raíz en i

$$\text{ALTURA}(T, n) \leq \text{ALTURA}(T, i)$$

Definición 3.2.0.3. La altura de un árbol T con centro en n se define como la distancia al nodo más lejano a n.

Para encontrar el nodo master se corre el algoritmo de búsqueda profunda (DFS) desde un nodo cualquiera. Este algoritmo itera por un grafo sin ciclos (en este caso dado un árbol) hasta la rama más larga. Una vez obtenida la rama más larga a partir de una raíz cualquiera, vamos a tomar a h, la hoja de esa rama. Luego, al árbol isomorfo T_h que tiene a la hoja h como raíz se aplica DFS a desde en nodo h y se obtiene la rama más larga desde h. Esta rama tiene la particularidad de ser la secuencia de nodos mas larga del árbol T. El servidor master será aquel que se encuentra en el medio de la secuencia. En caso de haber dos nodos que cumplan, se elige cualquiera.

Algoritmo 4 BusquedaMaster

Entrada:
G = (V,E)
color. Blanco;Gris;Negro
distancia: Entero
predecesor: Vertice
Salida:
master: Vertice

```
1: master ← Nil
2: mayor ← Nil
        ▷ Se inicializan vértices
3: para cada nodo v ∈ G.V hacer
4:     v.color ← Blanco
5:     v.predecesor ← Nil
        ▷ Se recorre grafo utilizando DFS,guardan distancia desde vértice v
6: para cada nodo v ∈ G.V hacer
7:     si v.color es Blanco entonces
8:         DFS(G,v)
        ▷ Busco el vértice a mayor distancia de v
9: para cada nodo v ∈ G.V hacer
10:    si mayor es Nil()∨_L v.tiempo > mayor.tiempo entonces
11:        mayor ← v
        ▷ Re-inicializo vértices
12: para cada nodo v ∈ G.V hacer
13:    v.color ← Blanco
14:    v.predecesor ← Nil
15:    v.distancia ← 0
        ▷ Ahora se recorre grafo utilizando DFS desde vértice mayor
16: DFS(g,mayor)
        ▷ Busco el vértice a mayor distancia del vértice encontrado en primera pasada
17: master ← Nil
18: para cada nodo v ∈ G.V hacer
19:    si mayor es Nil()∨_L v.distancia > mayor.distancia entonces
20:        mayor ← v
        ▷ El vértice que esta a la mitad entre mi raíz y el mas lejano es el master
21: master ← mayor
22: para i desde 1 hasta ⌊mayor.tiempo/2⌋ hacer
23:    master ← master.predecesor
24: retornar master
25: función DFS(G,v) v.color ← gris
26:    para cada nodo u ∈ G.Adj[v] hacer
27:        si u.color es Blanco( )entonces
28:            u.predecesor ← v
29:            u.distancia ← v.distancia + 1
30:            DFS(G,u)
31:    v.color ← negro
32: fin función
```

3.3 Justificación formal de correctitud

Consultora 1: selección de enlaces

El algoritmo de Kruskal es utilizado sin modificaciones. No demostraremos su correctitud en este informe. Para una demostración formal de correctitud ver [2, Cap.23.1]

Consultora 2: elección del maestro

Para justificar la correctitud de nuestro algoritmo tenemos que probar que:

1. obtener la secuencia de nodos mas larga de T resuelve el problema de obtener el nodo master.
2. las corridas de DFS sobre los árboles isomorfos T_n y T_h generan la secuencia más larga de T.

Definición 3.3.0.4.

1. T árbol generador mínimo del grafo G.
2. S secuencia más larga de nodos de T.
3. n tamaño de S.
4. R nodo que se encuentra a distancia $\lfloor n/2 \rfloor$ nodos del principio de S
5. T_r árbol isomorfo a T con raíz en R.
6. PRINCIPIO $= S[0...\lfloor n/2 \rfloor]$
7. FINAL $= S[\lfloor n/2 \rfloor ... n)$

Queremos probar que no existe otro árbol A isomorfo a T tal que

$$\text{RAÍZ}(A) <> N \wedge \text{ALTURA}(A) < \text{ALTURA}(T)$$

Primero veamos cual es la altura de T. Como la raíz de T es R, y R se encuentra a distancia $\lfloor n/2 \rfloor$ del principio de S, entonces R se encuentra como mínimo a $\lfloor n/2 \rfloor$ distancia de alguna de sus hojas. por lo tanto $\text{ALTURA}(t) \geq \lfloor n/2 \rfloor$.

Como S es la secuencia más larga de T, no existe una hoja en T_r que esté a más de $\lfloor n/2 \rfloor$ distancia de R porque T es conexo. Supongamos que existe una hoja h tal que h está a maá de $\lfloor n/2 \rfloor$ distancia de de R, entonces, como es T es conexo se puede llegar de cualquier nodo a cualquier otro. En particular se puede llegar a h desde R. Siendo que R tiene al menos dos hijos (principio y final), se podría formar una secuencia S' tal que esa rama se concatenaría con alguna de las ramas hijas de S (a la que no pertenezca), y esa secuencia sería más larga que S. Absurdo. Por lo tanto la altura de T_r es $\lfloor n/2 \rfloor$.

Ahora veamos que no existe un árbol isomorfo con T tal que su altura es menor a $\lfloor n/2 \rfloor$ y su raíz no es R. Es trivial ver que si R no es su raíz, R es un hijo. Como R tenía al menos dos hijos (principio y final), entonces el nodo adyacente a R en principio es padre de R o el nodo adyacente a R en final es padre de R u otro nodo. Si se toma el sub-árbol con raíz en el padre de R su altura es al menos la altura de principio o final +1. Como la altura de principio o final era $\lfloor n/2 \rfloor$, la altura del sub-árbol es mayor a la de T_r, por lo tanto no es un buen candidato para ser *master* y cualquier nodo padre en T de ese árbol va a tener altura mayor. Queda comprobado 1.

Ahora queremos probar que realizar las dos DFS nos devuelve la secuencia mas larga. Primero veamos como se comporta la DFS. La característica de la DFS es que va iterando hacia «abajo» en un árbol. Al contrario de BFS (que va descubriendo todos los nodos de un mismo nivel) DFS descubre primero el nodo más profundo de una rama. Lo que vamos a implementar es un contador en cada nodo, el cual cuenta la distancia a la raíz del árbol, tal que cada contador es igual al contador del padre +1. Luego tomamos el nodo con el contador máximo. Este nodo es el nodo mas lejano a la raíz.

Dada un nodo R' arbitrario de T, llamamos a $T_{r'}$ al árbol isomorfo de T con R' de raíz. Tomamos entonces el nodo mas lejano a R' en $T_{r'}$ aplicando DFS. Veamos que este nodo es un extremo de S. Como S es la secuencia más larga de T, $S \in T_{r'}$. Como T es conexo, cualquier nodo de S es alcanzable desde cualquier nodo de T. En particular, los extremos de S son alcanzables desde R' en $T_{r'}$. Como S es la secuencia máxima de nodos en T, R' está como máximo a n nodos del extremo más lejano de S. Se puede ver que si R' esta a $n-k$ nodos de distancia de un extremo de S, entonces R' esta a k nodos del otro extremo con $0 \leq k < n$. Si existiera un nodo que esté a mayor distancia que $\text{MÁX}(n-k, k)$ de R' se podría tomar la secuencia desde R' hasta ese nodo y formarse un secuencia (concatenada a la de mayor longitud de las que se están analizando) con longitud mayor a n. Por lo tanto, a través de una DFS desde cualquier nodo llego a alguno de los extremos de S.

Luego, el nodo mas lejano al extremo es el otro extremo de S ya que S era la secuencia más larga de nodos de T. Por lo tanto con un DFS y obteniendo el nodo más lejano queda definida la secuencia más larga. Así queda comprobado 2.

3.4 Cota de complejidad temporal

Consultora 1: selección de enlaces

El algoritmo de Kruskal se implementa sin modificaciones. Para ordenar las aristas se utiliza el procedimiento *make_heap* y para ordenarlos el procedimiento *sort_heap*. Ambos procedimientos son parte de la biblioteca estandar de C++. Por la documentación de la misma sabemos que la complejidad de los mismos es $\mathcal{O}(n)$ [10] y $\mathcal{O}(n * \log n)$ [11] respectivamente. Para implementar la estructura *disjoint-set-forest* [12] se utilizan las heurísticas *union by rank* [1, pág.178] y *path compression* [1, pág.179] las cuales permiten lograr un orden lineal, resultando una complejidad temporal del orden de $\mathcal{O}(E \log V)$ [2, pág.633] para el algoritmo de Kruskal.

[10] http://www.cplusplus.com/reference/algorithm/make_heap/
[11] http://www.cplusplus.com/reference/algorithm/sort_heap/
[12] Conjuntos disjuntos

Consultora 2: elección del maestro

La complejidad de la función está dada por los ciclos en los que itera y por el algoritmo DFS.

1. El primer ciclo itera por todos los nodos, inicializando su color en Blanco, y su predecesor en `null`. Estas incializaciones cuestan $\mathcal{O}(1)$, así que todo el ciclo tiene complejidad $\mathcal{O}(N)$.

2. *a*) Luego, se itera de nuevo por todos los nodos, llamando a la función DFS solo en los nodos que están pintandos de blanco.

   b) Por lo tanto el ciclo tiene coste $\mathcal{O}(N - 1 + 1 * (V + E))$. Entonces puedo acotar esta expresión por $N + N + N \in \mathcal{O}(N)$, el costo del ciclo.

3. La función DFS pregunta si el nodo por el cual entro es blanco y de serlo lo pinta de Gris. Luego se llama recursivamente a todos sus hijos (Blancos) con DFS, es decir, llama recursivamente a todos los nodos conexos con el nodo en el cual entro. Luego pinta al nodo inicial de Negro. Como el árbol que vamos a iterar es conexo, con una llamada a DFS con cualquier nodo inicial se van a pintar todos los nodos de negro al final del algoritmo. Es decir, para un nodo DFS va a tener costo $\mathcal{O}(V + E)$ y para todos los demás coste constante. Como el ciclo llama a DFS solo con los nodos que están blancos, despues de la primera llamada no se lo va a llamar más. Donde $N = V$, y como el grafo es un AGM $E < V$, en particular $E = V - 1$. Por lo tanto la función DFS tiene coste $N + N \in \mathcal{O}(N)$.

4. Luego se busca el vértice con distancia máxima, con una búsqueda lineal que como cuesta $\mathcal{O}(N)$, la complejidad no se ve afectada. Luego se vuelve a inicializar los nodos en costo $\mathcal{O}(N)$ como en 1.

5. Y ahora se realiza una DFS como en 3. Desde el nodo máximo con coste $\mathcal{O}(N)$.

6. Se itera por los nodos predecesores, desde $i = 0$ hasta $i = n/2$, es decir, $n/2$ operaciones de costo constante. Acoto esta complejidad por $\mathcal{O}(N)$, lo cual no afecta nuestra complejidad final.

Por lo tanto en estos 5 ciclos la complejidad se mantiene acotada por $\mathcal{O}(N)$, por lo tanto podemos acotar la complejidad final por $\mathcal{O}(N)$, con $N =$ Cantidad de vértices.

3.5 Verificación mediante casos de prueba

A fin de probar el correcto funcionamiento del algoritmo se realizaron pruebas con casos que consiramos representativos. Se acompaña cada *testcase* con un gráfico para facilitar la interpretación de la salida. Las aristas solución del primer problema se marcan con verde y el nodo master solución del segundo problema se marca con color rojo.

Testcase 1

Grafo trivial

```
Entrada:        Salida:
1 0             0 1
0
```

Figura 6: **Testcase 1**

Testcase 2

Grafo de 2 nodos con 1 arista. En este caso el problema de la selección del master tiene 2 soluciones óptimas. Nuestro algoritmo encuentra solo una de las soluciones óptimas poibles. Multiples corridas resultarán en misma seleccion de master.

```
Entrada:        Salida:
2 1             1 2 1 2
1 2 1
0
```

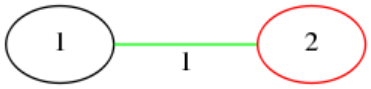

Figura 7: **Testcase 2**

Testcase 3

Grafo de 3 nodos. Solucion óptima única para ambos problemas.

```
Entrada:        Salida:
3 2             2 2 1 2 2 3
1 2 1
2 3 1
0
```

Figura 8: **Testcase 3**

Testcase 4

Grafo con 4 nodos y todas sus aristas de igual peso.

```
Entrada:        Salida:
4 3             3 2 1 2 2 3 3 4
1 2 1
2 3 1
3 4 1
0
```

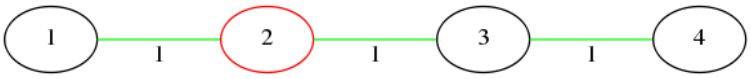

Figura 9: **Testcase 4**

Testcase 5

Grafo K_5 con todas sus aristas del mismo peso.

```
Entrada:        Salida:
5 10            4 1 1 2 1 3 1 4 1 5
1 2 1
1 3 1
1 4 1
1 5 1
2 3 1
2 4 1
2 5 1
3 4 1
3 5 1
4 5 1
0
```

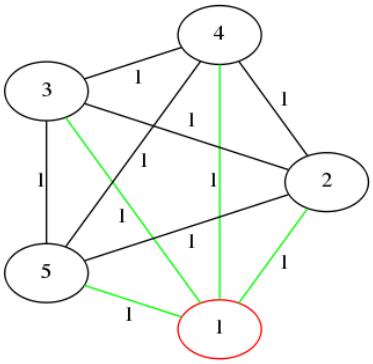

Figura 10: **Testcase 5**

Testcase 6

Grafo K_5 con aristas de diferente peso.

```
Entrada:        Salida:
5 8             4 1 1 2 1 3 1 4 1 5
1 2 1
1 3 1
1 4 1
1 5 1
2 3 5
3 4 5
4 5 5
5 2 5
0
```

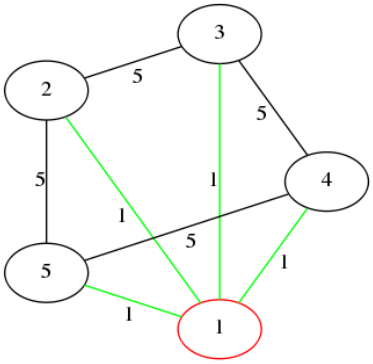

Figura 11: **Testcase 6**

Testcase 7

Similar a testcase 6, con diferentes pesos en las aristas.

```
Entrada:        Salida:
5 8             8 3 2 3 3 4 4 5 1 2
1 2 5
1 3 5
1 4 5
1 5 5
2 3 1
3 4 1
4 5 1
5 2 1
0
```

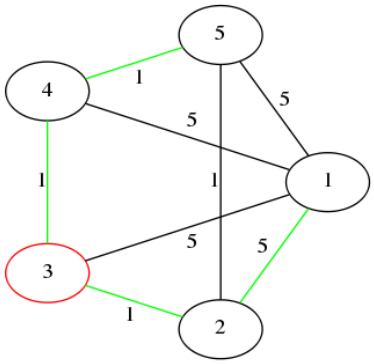

Figura 12: **Testcase 7**

3.6 Medición empírica de la performance

A fin de verificar la cota teórica de la solución provista por la Consultora 2 (elección del maestro), se realizaron múltiples mediciones sobre árboles generadores mínimos de 2 a 1000 vértices. Para generar la entrada se escribió un programa que genera grafos completos K_n y a cada una de las aristas se le asigna 1 valor random entre 1 y n.

Para cada entrada se ejecutan ambos algoritmos, el de la Consultora 1 y el de la Consultora 2 y se toma el tiempo del último. En el gráfico de la figura 19 se graficaron la cota teórica y el promedio de mediciones para cada tamaño de entrada. Como se puede observar el tiempo de ejecución crece linealmente como se había previsto.

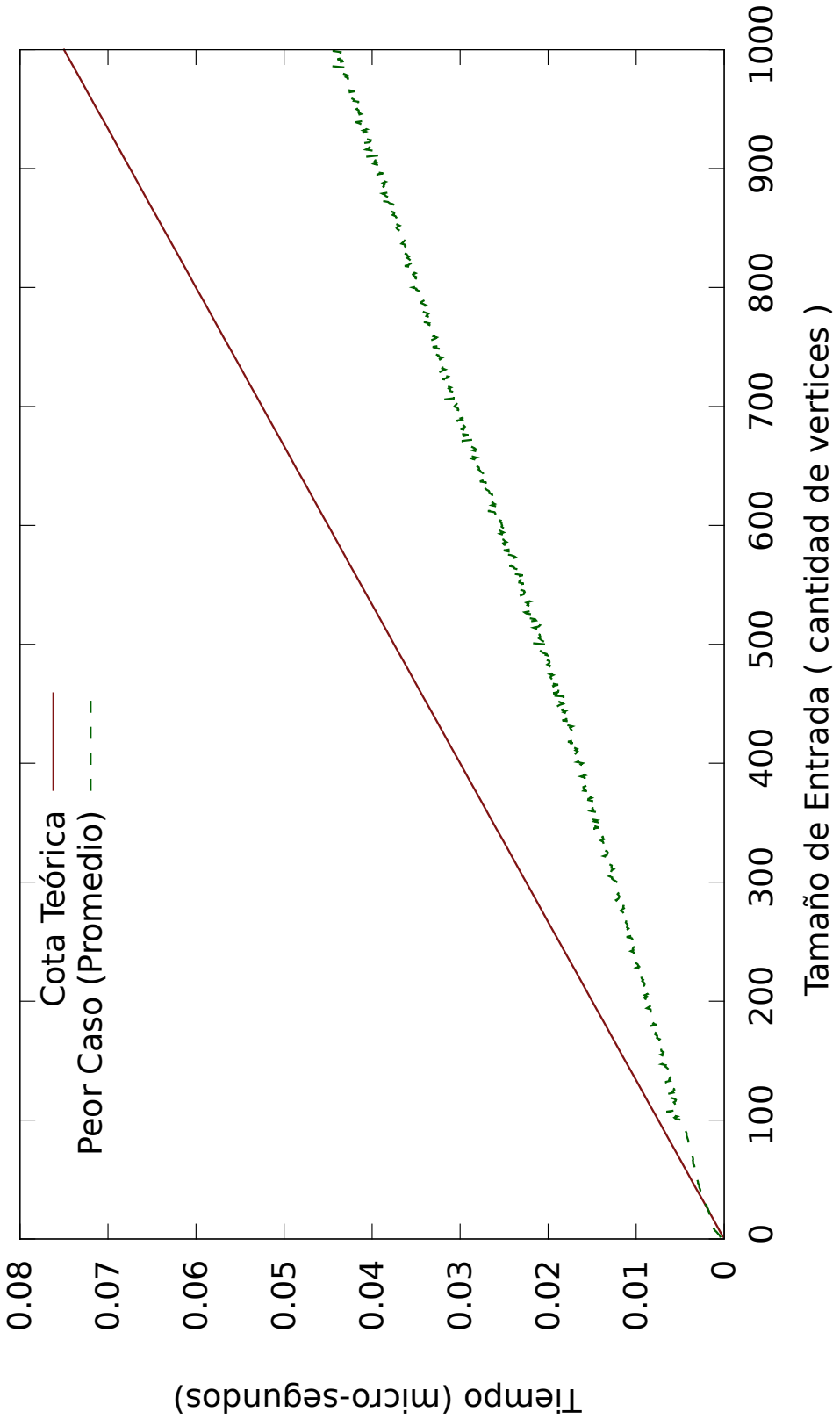

Figura 13: **Muestreo del Ejercicio 2**

3.7 Preguntas adicionales

Mostrar con un contraejemplo que es posible resolver las dos partes por separado de manera óptima pero que aun así haya una solución en la que la replicación termine en menos tiempo. Comentar posibles soluciones al problema.

Un buen ejemplo del caso en donde los dos problemas por separado darían una solución en donde la replicación termine en menor tiempo si se los considerara como un solo problema es la de un grafo fuertemente conexo (K_n) con todas las aristas de igual peso. Si por ejemplo tomamos el subgrafo que contiene a todos los nodos y donde cada nodo tiene exactamente dos aristas adyacentes y es conexo. Este grafo es un circuito simple, como lo que necesitamos es un árbol, podemos sacar cualquier arista y queda una arbol de altura N, donde $N = |V|$. Este árbol es un AGM y es solución del problema de la Consultora 1. Dada esta solución la replicacion mas corta que vamos a poder encontrar es de tiempo $N/2$ (altura del arbol /2). En cambio, si analizábamos los dos problemas como uno solo podíamos darnos cuenta que como el grafo era fuertemente conexo, con tomar el subgrafo con todos los vértices y todas las aristas adyacentes a un nodo, ese grafo tambien era AGM y su altura era 2. Por lo tanto la replicacion iba a terminar en tiempo constante.

Una posible solución para obtener la mejor solución sería obtener todos los AGM del grafo y de ese AGM, el de altura mínima, es decir, el que la secuencia más larga dentro del árbol sea la más corta de todos los posibles.

¿Cómo se debe modificar la solución si en lugar de transmitir por broadcast se lo hace por multicast, es decir, se debe mandar un paquete a cada destino, sin hacer copias?

La diferencia entre hacer un broadcast a un multicast sería que el costo de enviar un paquete desde un nodo n_1 a un nodo n_2 a través de un nodo n_3 en el broadcast era el costo de llegar desde n_1 a n_3 Y luego desde n_3 a n_2. En el multicast sería el costo de llegar desde n_1 a n_3 y luego el costo de llegar desde n_1 a n_2. Como se puede apreciar, las soluciones con estas dos formas van a ser distintas. Con el broadcast alcanzaba con encontrar el AGM del grafo de servidores, con el multicast habría que elegir el subgrafo formado por todos los vértices y las aristas que conecten al nodo que la sumatoria de todos los caminos al resto (server), sea mínima. Para encontrar este nodo habría que correr un algoritmo para encontrar el camino mínimo entre todos los nodos y luego seleccionar el que la suma de los caminos a los otros nodos fuera mínima.

4 Problema 3: Transportes pesados

4.1 Descripción

Motivación del ejercicio

Una empresa fabricante de ladrillos se encuentra en conflicto con la gobernación de una provincia a causa del daño gradual que están ocasionando los camiones de carga en las rutas a través de la que estos transportan la mercadería a cada uno de sus clientes. Para evitar que los caminos se sigan deteriorando, es necesario realizar una inversión para fortalecerlos. El costo total de esta inversión está regido por la candidad neta de Kilómetros de ruta que se fortalecerán.

Como sucede en la mayoría de los grupos empresariales, el directorio organizó una reunión a la que asistieron los inversores y, a pesar de la buena calidad de las masitas ninguno demostró, *motu proprio*, el más mínimo tipo de intención o interés en reparar **todas** las rutas, seguramente debido a la simple razón de que hacerlo les representaría una evidente disminución de la ganancia neta, sin aportar beneficios tangibles.

Luego de la correspondiente etapa de disputa de responsabilidades entre la empresa y el gobierno (en que nadie quería hacerse cargo de las inversiones) este último concluyó que a partir de cierta fecha las cargas de ladrillos solo podrían ser transportadas por las rutas que hayan sido debidamente fortalecidas (mediante una inversión por parte de la primera, obviamente).

Como no buscaban recurrir a la ilegalidad, y luego de evaluar la relación costo/beneficio del estresante trabajo que les correspondería si intentasen apelar esta última resolución del gobierno provincial, los directivos de la empresa acordaron en una segunda reunión (esta vez sin inversores presentes) que lo más conveniente era **respetar la decisión gubernamental** previendo que, ya que este mismo era el mensaje que iba a ser comunicado a los afectados por la decisión, esta también los exculpaba, vis maior, de verse en la responsabilidad de respaldarla.

Decidieron que, a pesar de haber decidido respetar decisión gubernamental [*o a causa de, según quién lo mire*], para mostrar y demostrar que la empresa se preocupaba por [*el dinero de*] sus inversores, se iba a encarar un proyecto de reorganización de las rutas que permitiera lograr una **máxima minimización de costos** al momento de fortalecerlas, aunque asegurando el **total abastecimiento de los actuales clientes**.

Descripción de los datos de entrada

Se brindarán los siguientes datos:

- Cantidad de fábricas (F)
- Cantidad de clientes (C)
- Cantidad de rutas (R)
- Composición y longitud de cada una de las R rutas

Objetivo a cumplir

Se encomienda la tarea de diseñar un algoritmo que, dada una distribución de rutas, junto con sus costos de reparación, permita encontrar la redistribución deseada, y que además se ejecute con una complejidad de $\mathcal{O}(C^2)$, o bien de $\mathcal{O}(R.log(C))$.

Formato de entrada

Utilizando la notación anterior, y siendo respectivamente s_n, f_n y l_n los puntos de inicio y fin, y longitud del camino n, la entrada se brindará bajo el siguiente formato:

F	C	R
s_1	f_1	l_1
s_2	f_2	l_2
...	...	...
s_{r-1}	f_{r-1}	l_{r-1}
s_r	f_r	l_r

Ejemplo de distribución de rutas

Dada la siguiente entrada:

```
3 6 12
1 4 50
1 3 10
1 6 1250
1 7 50
2 5 200
3 9 150
4 5 50
4 7 500
5 7 200
6 3 800
7 9 200
8 3 1
```

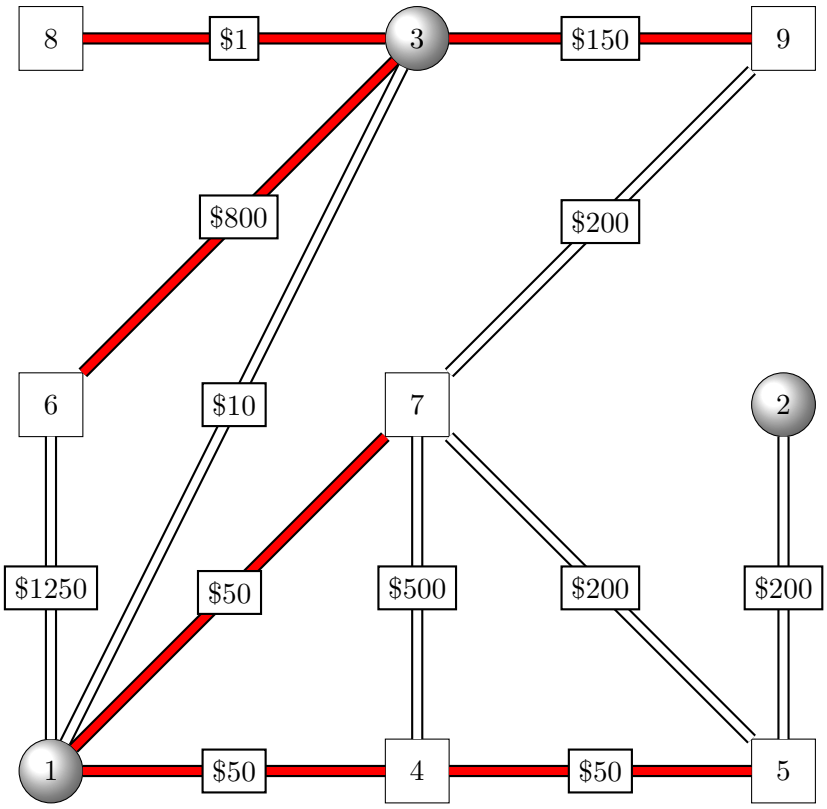

Figura 14: *Mapa provincial*

En *Figura 14* se volcó la representación gráfica de esta posible distribución de las rutas provinciales, en donde los círculos sombreados representan a cada una de las fábricas, cada uno de los nodos cuadrados indica un cliente, y cada arista está indicando una ruta junto con su costo de reparación. Se sombrearon en rojo, además, los caminos que forman la redistribución óptima de las rutas bajo las circunstancias expuestas en el gráfico.

4.2 Hipótesis de resolución

Dado el grafo $G = (V, E)$ donde cada vértice $v \in V$ tiene un color. Blanco para Cliente, negro para Fábrica. El conjunto de aristas son los caminos que puedan existir entre estos nodos. Se nos pide hallar el subgrafo $S = (V', E')$ inducido de G donde $\forall v,\ v \in V$, $\text{COLOR}(v) = \text{blanco} \implies v \in V'$ y existe un camino desde v a algún nodo $w \in V'$ tal que $\text{COLOR}(n) = \text{negro}$ y que no exista otra solución de este problema donde la sumatoria de pesos P de las aristas pertenecientes a E' sea mayor.

La forma que vamos a utilizar para resolver este problema es a traves del algoritmo de Kruskal de AGM. Lo que vamos a hacer es buscar un escenario donde Kruskal pueda funcionar y nos devuelva una posible solución de lo que planteamos. Se puede ver que:

- El subgrafo que nos piden no es necesariamente conexo. Podría haber hasta k subgrafos que no se conecten entre si.
- Una vez que establezco un subgrafo s de G conexo y de costo mínimo, ese subgrafo como mucho tiene conectada una fábrica, ya que desde esa fábrica puede llegar a cualquier cliente de s. Entonces, si la solucion s' es un subgrafo de G que se divide en k subgrafos no conexos entre si, s' tiene exactamente k fábricas.
- Una arista con peso p, $p > 0$ que une dos fábricas, nunca pertenece a la solución. Esto es porque conectar dos fábricas no aporta nada ya que cualquier cliente que se conecte a alguna de esas fábricas no necesita llegar a la otra por esta arista porque puede abastecerse de la primera.
- Cada subgrafo disjunto de la solución es un árbol con exactamente una fábrica. Por lo tanto la solución es un bosque.

Dado este grafo, la primera intuición de solución sería encontrar un AGM de G. El problema es que este AGM podría tener aristas de más, ya que hay aristas «descartables».

Solución

A partir del grafo G descrito anteriormente vamos a crear el grafo $G' = (V', E')$ tal que G está incluido en G' y existe el nodo $f \in V'$, $f \notin V$ tal que $\forall v,\ v \in V \iff \text{COLOR}(v) = \text{negro}$. Entonces $(f, v) \in E'$ y $P(f, v) = 0$. Es decir, agregamos un nodo a G, el cual está conectado

a todas las fábricas con aristas de peso 0. Luego vamos a encotrar el AGM de G'. Finalmente vamos a remover las aristas que agregamos anteriormente.

Algoritmo 5 Selección de Rutas

Entrada:
G = (V,E)
Salida:
A: conjunto de aristas del AGM

```
1:  G.E ← G.E ∪{nodo_virtual}
2:  para cada nodo tipo fabrica v ∈ G.V hacer
3:      Agregar arista de peso 0 (v,nodo_virtual) a G.E
4:  A ← AGM-Kruskal(G)
5:  para cada arista (u,v) ∈ A hacer
6:      si arista incide en nodo_virtual entonces
7:          A ← A\{arista}
8:  retornar A
9:  función AGM-Kruskal(G)
        ▷ Los siguiente arreglos se ultizan para implementar Conjuntos Disjuntos
    height[1...N]: Vector de alturas
    set[1...N]: Vector de padres
10:     A = ∅
11:     para cada nodo v ∈ G.V hacer
12:         height[v] ← 0
13:         set[v] ← v
14:     OrdenarPorPeso(G.E)
15:     para cada arista (u,v) ∈ G.E,tomadas en orden creciente por peso hacer
16:         si DevolverConjunto(u) <> DevolverConjunto(v) entonces
17:             A = A ∪ {(u, v)}
18:             UnirConjuntos(u,v)
19: fin función
20: función DevolverConjunto(v)
21:     r ← v
22:     mientras set[r] ≠ r hacer
23:         r ← set[r]
24:     i ← v
25:     mientras i ≠ r hacer
26:         j ← set[i]
27:         set[i] ← r
28:         i ← j
29:     retornar r
30: fin función
31: función UnirConjuntos(u,v)
32:     si height[u] = height[v] entonces
33:         height[u] ← height[u] + 1
34:         set[v] ← u
35:     sino
36:         si height[u] > height[v] entonces
37:             set[v] ← u
38:         sino
39:             set[u] ← v
40: fin función
```

4.3 Justificación formal de correctitud

El grafo S solución de G es un bosque de k árboles y k fábricas donde cada árbol tiene conectada exactamente 1 fábrica y el peso del grafo es mínimo. Supongamos que todos los vertices de G están incluidos en S. Se podrían descartar las fábricas que no están conectadas a ningún cliente, pero como son árboles y no tienen peso, cumplen con las restricciones y no van a influir en la solución. Entonces S lo único que recorta son aristas de G.

Se puede ver entonces que agregar un nodo r conectado a todas las fábricas con aristas de peso 0 tampoco afectaría la solución. La conversión de un grafo a otro es a través de descartar ese nodo y todas sus aristas adyacentes. El peso de la solución y ese grafo inducido es el mismo. Una vez que agregamos este nodo, ahora la solución es un árbol. Si tomamos de raíz a r, este árbol tiene k ramas, donde k eran los árboles de S. Llamemos S_r a este grafo. Llamemos G_r al grafo construido a partir de G con el mismo procedimiento. Es decir, agregar un nodo y conectar todas las fábricas a ese nodo con aristas de peso 0.

- Sea $G = (V, E)$ el grafo original.
- Sea $S = (V, E_s)$ el grafo solucion de G.
- Sea r un nodo, $r \notin G$.
- Sea A un conjunto de aristas tal que

$$\forall v : nodo,\ v \in V \land \text{COLOR}(v) = \text{negro} \iff (r, v) \in A \land P((r, v)) = 0$$

- Sea $G' = (V', E')$ un grafo tal que $V' = V \cup \{r\} \land E' = E \cup A$.
- Sea $S' = (V', E'_s)$ un grafo tal que $V' = V \cup \{r\} \land E_{s'} = E_s \cup A$.

Para justificar este algoritmo hay que probar que:

1. Agregar o quitar un nodo con aristas de peso 0 a G y luego descartarlas no influye en la solución.
2. S' es un AGM de G'.
3. S' es un AGM de $G' \implies S$ es una solución de G.

Primero deberíamos ver que el conjunto A tiene peso 0, ya que todas las aristas son de peso 0. Si agrego una arista de peso 0 a G va a estar incluída en la solución ya que no afecta el peso total (luego la podemos descartar). Por lo tanto, agregar el conjunto A a la solución no va a aumentar su peso. Lo que sí podría pasar es que se pise alguna arista de S, pero esto no es posible porque A sólo agrega aristas adyacentes a un nodo que también agregamos. Por lo tanto se puede ir desde G a G' y de S a S' por el mismo procedimiento y no influye en la solución. Así queda comprobado 1.

S es el grafo solución de G, por lo tanto los nodos de S son los mismos que los de G. S está dividido en K subgrafos conexos tal que cada subgrafo está conectado a exactamente 1 fábrica y S es de costo mínimo. Los nodos de G' son los mismos que los de S'. Como todas las

fábricas de S' están conectadas con un arista de peso 0 a r y los subgrafos estaban conectados a exactamente a una fábrica, S' es un árbol. Como S era de costo mínimo y todas las aristas que agregue en S' son de costo 0, S' es de costo mínimo. Entonce S' es un árbol de costo mínimo (conexo) que contiene a todos los nodos de G', por lo tanto es un AGM de G'. Una justificación mas simple es que S' fue obtenido por Kruskal a partir de G', así que es un AGM. Así queda comprobado 2.

Por definicion, G' contiene a G. Ahora suponiendo que ya se encontró S'; G' y S' es un AGM de G'. G está contenido en G', por lo tanto los nodos y las aristas de G, están contenidas en el conjunto de nodos y de aristas de G'. Como S' es un AGM de G', S' es conexo y de costo mínimo. Como las aristas de A son de peso 0, todas las aristas de A están incluidas en S'. Podrían no estarlo si existe otra arista de peso 0, pero supongamos que descartamos ese caso ya que siempre pertenece a la solución.

Como las aristas de A van solo desde r a una fábrica, y S' es un AGM, los subgrafos adyacentes a r son disjuntos pero conexos cada uno. Si le saco el nodo r y todas sus aristas adyacentes, quedan k subgrafos disjuntos con exactamente una fábrica conectada donde $k = |A|$ (ya que las aristas de r solo conectaban fábricas). Como todas las fábricas de G' estaban conectadas a r, todos los subgrafos adyacentes a r contienen exactamente una fábrica, ya que que una arista de peso mayor que 0 que conecte a dos subgrafos perdería contra las aristas que agregué de peso 0. Es decir, el camino a través de r que construí sería menos pesado. Como S' contenía a todos los nodos de G' y solo se sacó el nodo r, esta solución contiene a todos los nodos de G. Por lo tanto, todos los nodos están conectados a al menos a una fábrica.

Como las aristas de A eran de costo 0, esta solución sigue siendo de costo mínimo. Entonces quedaron todos los nodos de G conectados a exactamente una fábrica y con costo mínimo, lo que representa la definición de S. Asi queda comprobado 3.

4.4 Cota de complejidad temporal

Primero agregamos un nodo al grafo, que consiste en sumar 1 a la cantidad de vértices guardada por la estructura Grafo. Luego, por cada fábrica agregamos una arista entre la misma y el nodo virtual. Las aristas se guardan en un contenedor de tipo vector de la biblioteca estandar de C++. Como la memoria para las aristas extra se reserva con anterioriadad, agregar una nueva arista es $\mathcal{O}(1)$ y por lo tanto agregar F aristas es $\mathcal{O}(F)$.

A este Grafo modificado se le aplica el algoritmo de Kruskal, utilizando las mismas estructuras y heurísticas que en el ejercicio anterior, logrando una complejidad $\mathcal{O}(m \log n)$, siendo $n = |V|$ y $m = |E|$. Al conjunto de aristas A le removemos las aristas de peso 0 que se agregaron en el segundo paso. Por documentación de la biblioteca standard de C++, la complejidad de esta operación es $\mathcal{O}(\text{MÁX}(F, m))$, lineal en la cantidad de elementos borrados más lineal en la cantidad de elementos que se encuentren a continuación de los borrados.

Luego, el orden del algoritmo va a estar dado por $\text{MÁX}(F, m \log n, \text{MÁX}(F, m)) \in \mathcal{O}(m \log n)$. Además, sabemos que el número de fábricas nunca es mayor al número de clientes, por lo que $m \log n \leq m \log(2 * C)$. Por propiedades logarítmicas, $m \log(2 * C) = m * (\log 2 + \log C)$, por lo que se puede asegurar que algoritmo tiene una complejidad $\mathcal{O}(m \log C)$, como era requerido por el enunciado.

4.5 Verificación mediante casos de prueba

Para verificar el correcto funcionamiento de nuestro programa tomamos una serie de instancias que consideramos representativas de casos extremos o sensibles y analizamos el comportamiento. Definimos los siguientes casos:

Testcase 1 Configuración con multiples soluciones. Nuestro algoritmo encuentra una solución de todas las soluciones validas. Multiples ejecuciones devuelven siempre el mismo resultado.

```
Entrada:          Salida:
3 4 7             8 4 5 5 6 6 7 1 4
1 4 5
2 5 5
3 6 5
1 7 5
4 5 1
5 6 1
6 7 1
0
```

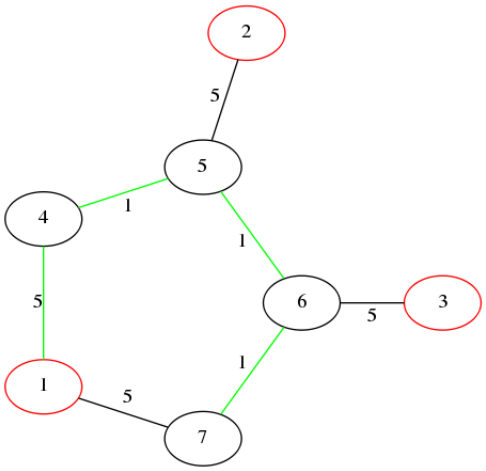

Figura 15: **Testcase 1**

Testcase 2 Grafo con todas las aristas del mismo peso.

```
Entrada:          Salida:
3 4 7             4 1 4 1 7 2 5 3 6
1 4 1
2 5 1
3 6 1
1 7 1
```

```
4 5 1
5 6 1
6 7 1
0
```

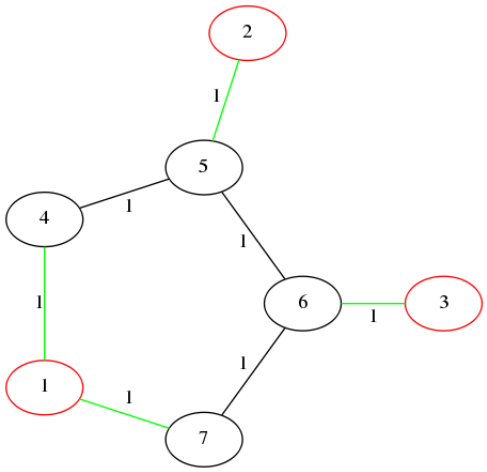

Figura 16: **Testcase 2**

Testcase 3 Grafo con única solución óptima.

```
Entrada:          Salida:
3 3 9             3 1 4 4 5 4 6
1 4 1
2 4 2
3 4 3
4 5 1
4 6 1
4 7 1
1 2 1
1 3 1
2 3 1
0
```

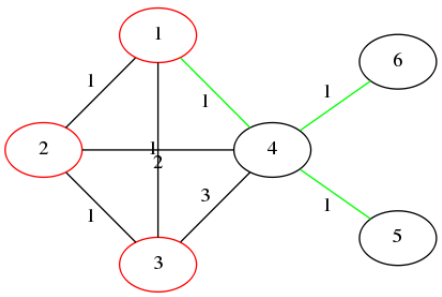

Figura 17: **Testcase 3**

Testcase 4 Grafo con clientes aislados.

```
Entrada:          Salida:
3 3 6             3 1 4 2 5 3 6
1 4 1
2 5 1
3 6 1
1 2 1
1 3 1
2 3 1
0
```

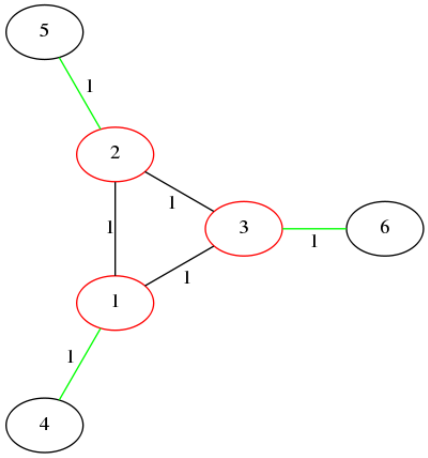

Figura 18: **Testcase 4**

4.6 Medición empírica de la performance

A fin de verificar la cota teórica de la solución provista se realizaron múltiples test sobre grafos completos K_n, con $2 \leq n \leq 500$. A las aristas del grafo se le asigno un peso random entre 1 y n. De los n nodos de cada grafo, 40 % eran fábricas y 60 % clientes, como se ve en la figura 19.

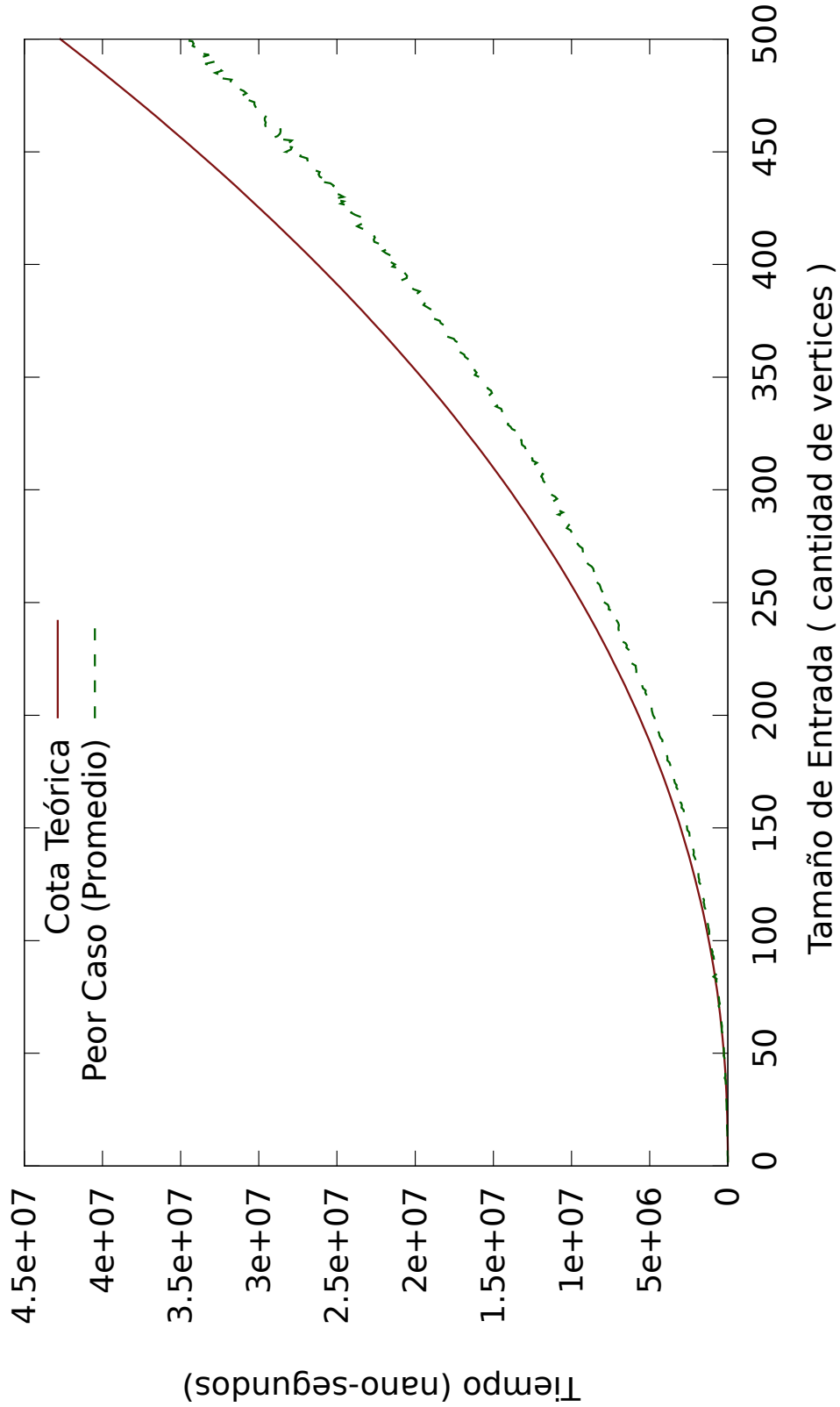

Figura 19: **Muestreo del Ejercicio 3 - Teórica vs Muestra empírica**

Además, se realizaron múltiples mediciones variando el porcentaje de fábricas. Como se

ve en la figura 20, la performance del algoritmo no varía entre las diferentes mediciones. En la figura solo se muestran mediciones para 10 % y 40 % de fábricas. Se omiten el resto de las mediciones dado que no aportan información.

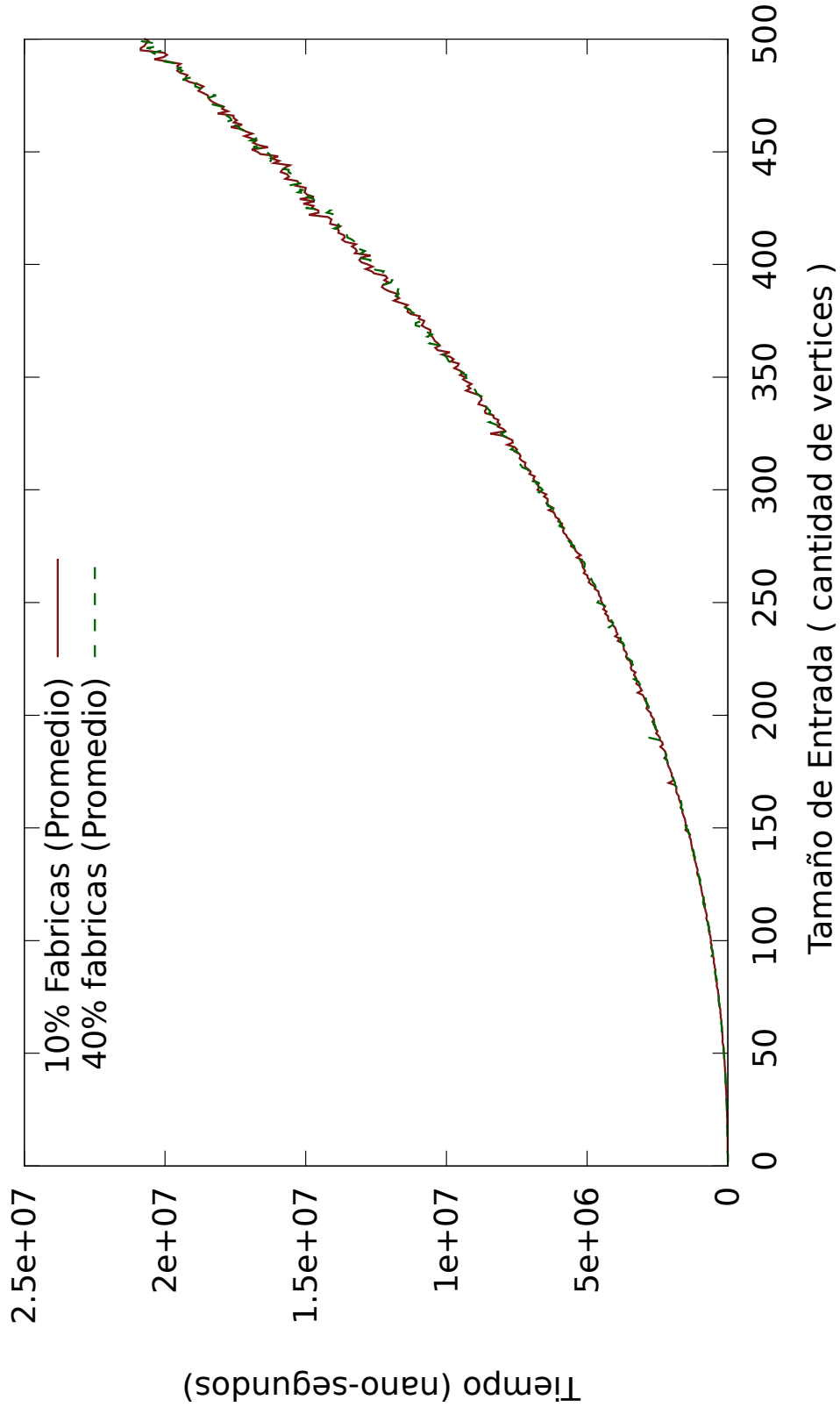

Figura 20: **Muestreo del Ejercicio 3 - Variación porcentaje fabricas**

5 Apéndices

5.1 Informe de Modificaciones

Problema 1: Impresiones ordenadas

- Se amplió la descriptividad en torno a la *Ecuación* (1) de manera de clarificar lo que pretende calcular.
- Se corrigieron aspectos formales en la hipótesis.
- Se modificó el pseudocódigo para que quede menos implementativo.
- Se cambió por completo la justificación formal de correctitud.
- Se corrijieron aspectos del análisis de complejidad de manera de hacerlo más conciso y preciso.
- Se agregaron mediciones empíricas de performance.

Problema 2: Replicación de contenido

- Se agregaron referencia faltantes en la hipótesis de solución y en calculo de complejidad teórica.
- Se agregaron test cases para prueba empirica de correctitud.
- Se agrandó el tamaño de entrada de mediciones de performance y se completo el analisis.
- Se agregó código de programa generador de grafos para tests empiricos.

Problema 3: Transportes pesados

- Se modificó hipótesis de solución.
- Se modificó la Justificación formal.
- Se agregaron test cases para prueba empirica de correctitud.
- Se agregaron mediciones de performance
- Se modificó y completo el código.
- Se agregó código de programa generador de grafos para tests empiricos.

5.2 Código Fuente (resumen)

Problema 1: Impresiones ordenadas

Listing 1: ej1.cpp

```
#include "ej1.h"
int main( int argc, char** argv ){
  ParserDeParametros parser( argc, argv );
  if( ! parser . parametrosSonValidos() ) {
    parser . imprimirAyuda();
    return -1;
  }
  Ejercicio1 ejercicio = Ejercicio1();
  while( ejercicio.obtenerInput( parser.input() ) ){
    ejercicio.resolver();
    ejercicio.imprimirOutput( parser.output() );
  }
  return 0;
}
```

Listing 2: Ejercicio1.cpp

```
#include "Ejercicio1.h"
bool Ejercicio1::obtenerInput(istream &input){
  // Si ya paso algun testcase, "Reinicio" los contenedores
  A.clear();
  unaCola.clear();
  cantTrabajos  = 0;
  costoTotal    = 0;
  cantEncolados = 0;
  int testcase;
  input >> testcase;                          // Primer parametro
  if(testcase != TESTCASE_NULL){
    cantTrabajos = testcase;
    A.reserve(cantTrabajos+1);
    unaCola.reserve(cantTrabajos);
    vector<int> fila (cantTrabajos+1, -1);
    for(int i = 0; i <= cantTrabajos; i++)  // lleno la matriz de nulos
      A.push_back(fila);
    for(int i = 1; i <= cantTrabajos; i++){ // cargo la matriz
      for(int j = 0; j < i; j++){
        input >> A[j][i];
      }
    }
  } else return false;  // Testcase == TESTCASE_NULL
  return true;
}

void Ejercicio1::resolver( void ){
  calcularMinimoPresupuesto();
  generarColaDeMinimoPresupuesto();
}

void Ejercicio1::imprimirOutput( ostream &output ){
  output << costoTotal;
  output << " " << cantEncolados;
```

```
35    imprimirCola(output);
36    output << endl;
37  }
38
39  void Ejercicio1::calcularMinimoPresupuesto(){
40    int min = INF;
41    int presu = INF;
42    for(int j = cantTrabajos - 1; j >= 0; j--){
43      presu = mejorPresupuesto(cantTrabajos, j);
44      A[cantTrabajos][j] = presu;
45      if(presu < min)
46        min = presu;
47    }
48    costoTotal = min;
49  }
50
51  int Ejercicio1::mejorPresupuesto(int i, int j){
52    int value;
53    if( (i == 0 && j == 0) || (i <= j) )
54      value = INF;
55    else if(i == 1)
56      value = costo(0, 1);
57    else if(i > j + 1)
58      value = presupuesto(i-1, j) + costo(i-1, i);
59    else if(i == j + 1)
60      value = minimoMejorPresupuestoHastaElTrabajo(i);
61    else
62      value = INF;
63    return value;
64  }
65
66  int Ejercicio1::minimoMejorPresupuestoHastaElTrabajo(int i){
67    return minimoDeFila(i);
68  }
69
70  int Ejercicio1::minimoDeFila(int i, bool indice){
71    int min = INF;
72    int imin = INF;
73    int presu = INF;
74    for(int k = i - 2; k >= 0; k--){
75      presu = presupuesto(i-1, k) + costo(k,i);
76      if(presu < min){
77        min = presu;
78        imin = k;
79      }
80    }
81    return indice ? imin : min;
82  }
83
84  void Ejercicio1::generarColaDeMinimoPresupuesto(){
85    vector< pair<int,int> > listaIndices;
86    listaIndices.reserve(cantTrabajos);
87    int n = cantTrabajos;
88    int m = minimoIndice(n);
89    listaDeIndices(listaIndices, n, m);
90    int iPrev = 0;
91    for(vector< pair<int,int> >::reverse_iterator rit = listaIndices.rbegin()
          ; rit != listaIndices.rend(); ++rit){
92      if((*rit).first == iPrev){
93        unaCola.push_back((*rit).second);
94        iPrev = (*rit).second;
95      }
96    }
```

```
97    cantEncolados = unaCola.size();
98  }
99
100 vector< pair<int,int> > Ejercicio1::listaDeIndices(vector< pair<int,int> >
        &lista, int i, int j){
101   pair<int,int> iCosto (-1,-1);
102   if(i == 1 && j == 0){
103     iCosto.first = 0;
104     iCosto.second = 1;
105     lista.push_back(iCosto);
106     return lista;
107   }
108   if(j < i - 1){
109     iCosto.first = i-1;
110     iCosto.second = i;
111     lista.push_back(iCosto);
112     return listaDeIndices(lista, i-1, j);
113   }
114   if(j == i -1){
115     int k = minimoDeFila(i, true);
116     iCosto.first = k;
117     iCosto.second = i;
118     lista.push_back(iCosto);
119     return listaDeIndices(lista, j, k);
120   }
121   return lista;
122 }
123
124 int Ejercicio1::minimoIndice(int i){
125   int min = INF;
126   int presu = INF;
127   int indice = INF;
128   for(int j = 0; j < i; j++){
129     presu = presupuesto(i, j);
130     if(presu < min){
131       min = presu;
132       indice = j;
133     }
134   }
135   return indice;
136 }
137
138 int Ejercicio1::presupuesto(int i, int j){
139   if(A[i][j] == NULO)
140     A[i][j] = mejorPresupuesto(i, j);
141   return A[i][j];
142 }
143
144 int Ejercicio1::costo(int j, int i){
145   return A[j][i];
146 }
147
148 void Ejercicio1::imprimirCola( ostream &output ){
149   for (vector<int>::iterator it = unaCola.begin(); it != unaCola.end(); it
        ++)
150     output << " " << *it;
151 }
```

Problema 2: Replicación de contenido

Listing 3: ej2.cpp

```
#include "ej2.h"
int main(int argc,char **argv){
  Grafo g;
  VectorAristas a;
  int master;


  while(true){
    a.clear();
    if(!cargarGrafo(g))
      break;
    AGMKruskal(g,a);
    master = BusquedaMaster(g);
    ImprimirResultado(a,master);
  }

  return 0;
}

void AGMKruskal(Grafo &g,VectorAristas &a){
  //Arreglos para implementar disjoint sets
  vector<int> height(g.cantidad_vertices,0);
  vector<int> set(g.cantidad_vertices,0);
  //Inicializo cada nodo como un set disjunto
  for(int i = 0;i< g.cantidad_vertices;i++) set[i] = i ;
  //Ordeno Aristas por peso
  make_heap(g.vector_aristas.begin(),g.vector_aristas.end());
  sort_heap(g.vector_aristas.begin(),g.vector_aristas.end());
  for( unsigned int i = 0; i < g.vector_aristas.size(); i++){
    int set_u = DevolverConjunto(set,g.vector_aristas[i].second.first);
    int set_v = DevolverConjunto(set,g.vector_aristas[i].second.second);
    if(  set_u != set_v){
      UnirConjuntos(set,height,set_u,set_v);
      a.push_back(g.vector_aristas[i]);
    }
  }
  //Completo listas de adyacencias para parte b
  for( unsigned int i = 0 ; i < a.size();i++){
    Vertice u = a[i].second.first;
    Vertice v = a[i].second.second;
    g.adyacencias[u].push_back(v);
    g.adyacencias[v].push_back(u);
  }
}

Vertice BusquedaMaster(Grafo &g){
  VerticeDFS master,mayor,raiz;
  int mayor_index = 0;
  int master_index,raiz_index;

  for( unsigned int i = 0; i<g.vertices.size();i++) {
    g.vertices[i].color = BLANCO;
    g.vertices[i].distancia = 0;
  }

  for( unsigned int i = 0; i<g.vertices.size();i++) {
```

```
57      if( g.vertices[i].color == BLANCO )
58        DFS(g,i);
59    }
60
61    /**
62     * Busco en timepo lineal el nodo que mas lejos esta de mis raiz.
63     */
64    mayor = g.vertices[0];
65    for( unsigned int i = 1; i < g.vertices.size();i++){
66      if( g.vertices[i].distancia > mayor.distancia ){
67        mayor = g.vertices[i];
68        mayor_index = i;
69      }
70    }
71
72    for( unsigned int i = 0; i<g.vertices.size();i++) {
73      g.vertices[i].color = BLANCO;
74      g.vertices[i].distancia = 0;
75    }
76
77    raiz_index = mayor_index;
78    raiz = mayor;
79    DFS(g,raiz_index);
80
81    mayor = g.vertices[0];
82    for( unsigned int i = 1; i < g.vertices.size();i++){
83      if( g.vertices[i].distancia > mayor.distancia ){
84        mayor = g.vertices[i];
85        mayor_index = i;
86      }
87    }
88
89    master = mayor;
90    master_index = mayor_index;
91    for( int i = 1; i<=( mayor.distancia /2 );i++){
92      master_index = master.predecesor;
93      master = g.vertices[master.predecesor];
94    }
95    return master_index;
96  }
97
98  void DFS(Grafo &g,Vertice v){
99    g.vertices[v].color = GRIS;
100   vector<int>::iterator itr;
101
102   itr = g.adyacencias[v].begin();
103
104   while( itr != g.adyacencias[v].end() ){
105     if( g.vertices[(*itr)].color == BLANCO ){
106       g.vertices[(*itr)].predecesor =v;
107       g.vertices[(*itr)].distancia = g.vertices[v].distancia +1;
108       DFS(g,(*itr));
109       }
110     itr++;
111   }
112   g.vertices[v].color = NEGRO;
113 }
114
115 bool cargarGrafo(Grafo &g){
116   int peso,u,v;
117   g.vector_aristas.clear();
118   g.adyacencias.clear();
119   cin >> g.cantidad_vertices;
```

```
120   if( g.cantidad_vertices == 0 )
121     return false;
122   g.vertices.resize(g.cantidad_vertices);
123   cin >> g.cantidad_aristas;
124   g.adyacencias.resize(g.cantidad_vertices);
125   for( int i = 1 ; i <= g.cantidad_aristas;i++){
126     cin >> u;
127     cin >> v;
128     cin >> peso;
129     u--,v--;
130     g.vector_aristas.push_back(make_pair(peso,make_pair(u,v)));
131   }
132   for( int i = 0; i<g.cantidad_vertices;i++) {
133     g.vertices[i].color = BLANCO;
134     g.vertices[i].distancia = 0;
135   }
136   return true;
137 }
138
139 int DevolverConjunto(vector<int> &set,int u){
140   static int i,j;
141   int r = u;
142   while( set[r] != r ) r = set[r];
143   i = u;
144   while( i != r ){
145     j = set[i];
146     set[i] = r;
147     i = j;
148     set[i] = r;
149     i = j;
150   }
151   return r;
152 }
153
154 void UnirConjuntos(vector<int> &set,vector<int> &height,int u,int v){
155   if( height[u] == height[v] ){
156     height[u]++;
157     set[v] = u;
158   }else{
159     if( height[u] > height[v] )
160       set[v] = u;
161     else set[u] = v;
162   }
163 }
164
165
166 void ImprimirResultado(VectorAristas &a,Vertice master){
167   int costo = 0;
168   for( unsigned int i = 0; i < a.size();i++ ){
169     costo += a[i].first;
170   }
171   cout << costo << " " << master + 1<< " ";
172   for( unsigned int i = 0; i < a.size();i++ )
173     cout << a[i].second.first + 1 << " " << a[i].second.second + 1<< " ";
174   cout << endl;
175 }
```

Listing 4: ej2.h

```
1 typedef int Vertice;
2 typedef pair<int,pair<Vertice,Vertice> > Arista;
3 typedef vector<Arista> VectorAristas;
```

```
4  typedef vector< vector<int> > VectorAdyacencias;
5  enum ColorNodo{BLANCO,GRIS,NEGRO};
6
7  typedef struct VerticeDFS_{
8    ColorNodo color;
9    int distancia;
10   Vertice predecesor;
11 } VerticeDFS;
12
13 typedef struct {
14   int cantidad_vertices;
15   int cantidad_aristas;
16   vector<VerticeDFS> vertices;
17   VectorAdyacencias adyacencias;
18   VectorAristas vector_aristas;
19 } Grafo;
20
21 bool cargarGrafo(Grafo &g);
22 void ImprimirResultado(VectorAristas &a,Vertice master);
23 void AGMKruskal(Grafo &g,VectorAristas &a);
24 int DevolverConjunto(vector<int> &set,Vertice u);
25 void UnirConjuntos(vector<int> &set,vector<int> &height,Vertice u,Vertice v
       );
26 int BusquedaMaster(Grafo &g);
27 void DFS(Grafo &g,Vertice v);
```

Listing 5: random_graph.cpp

```
1  int main(int argc,char **argv){
2    if(argc < 3){
3      cout << "Usage: " << argv[0] << " <max_n> <max_weight> <%fabricas> " <<
           endl;
4      exit(1);
5    }
6    int max_n = atoi(argv[1]);
7    int max_w = atoi(argv[2]);
8    int aristas = 0;
9    srand(time(NULL));
10   cout <<  max_n << " " << ((max_n * (max_n -1 )) /2 )<< endl;
11   for( int i = 1; i <= max_n; i++)
12     for( int j = i + 1; j <= max_n;j++){
13       cout << i << " " << j << " " << ( rand() % max_w ) + 1 << endl;
14     }
15   cout << "0" << endl;
16   return 0;
17 }
```

Problema 3: Transportes pesados

Listing 6: ej3.cpp

```cpp
#include "ej3.h"
int main(int argc,char **argv)
{
  Grafo g;
  VectorAristas a;
  int fabricas;
  while(true){
    a.clear();
    if( ( fabricas = cargarGrafo(g) ) == 0)
      break;
    AGMKruskal(g,a);
    ImprimirResultado(a,g.cantidad_vertices);
  }
  return 0;
}

void AGMKruskal(Grafo &g,VectorAristas &a){
  //Arreglos para implementar disjoint sets
  vector<int> height(g.cantidad_vertices,0);
  vector<int> set(g.cantidad_vertices,0);
  //Inicializo cada nodo como un set disjunto
  for(int i = 0;i< g.cantidad_vertices;i++) set[i] = i ;
  //Ordeno Aristas por peso
  make_heap(g.vector_aristas.begin(),g.vector_aristas.end());
  sort_heap(g.vector_aristas.begin(),g.vector_aristas.end());
  for( unsigned int i = 0; i < g.vector_aristas.size(); i++){
    int set_u = DevolverConjunto(set,g.vector_aristas[i].second.first);
    int set_v = DevolverConjunto(set,g.vector_aristas[i].second.second);
    if(  set_u != set_v){
      UnirConjuntos(set,height,set_u,set_v);
      a.push_back(g.vector_aristas[i]);
    }
  }
  //Completo listas de adyacencias para parte b
  for( unsigned int i = 0 ; i < a.size();i++){
    Vertice u = a[i].second.first;
    Vertice v = a[i].second.second;
    g.adyacencias[u].push_back(v);
    g.adyacencias[v].push_back(u);
  }
}

int cargarGrafo(Grafo &g){
  int peso,u,v;
  int fabricas,clientes;
  g.vector_aristas.clear();
  g.adyacencias.clear();
  //Leo Cantidad de fabricas
  cin >> fabricas;
  if( fabricas == 0 )
    return 0;
  //Leo Cantidad de Clientes
  cin >> clientes;
  g.cantidad_vertices = fabricas + clientes;
  g.vertices.resize(g.cantidad_vertices);
  cin >> g.cantidad_aristas;
```

```
57    g.adyacencias.resize(g.cantidad_vertices +  fabricas);
58    for( int i = 1 ; i <= g.cantidad_aristas;i++){
59      cin >> u;
60      cin >> v;
61      cin >> peso;
62      u--,v--;
63      g.vector_aristas.push_back(make_pair(peso,make_pair(u,v)));
64    }
65    // Agrego nodo magico ;)
66    g.cantidad_vertices++;
67    //Agrego aristas de peso cero entre fabricas y nodo magico
68    for( int i = 0 ; i < fabricas ;i++){
69      g.cantidad_aristas++;
70      g.vector_aristas.push_back(make_pair(0,make_pair(i,g.cantidad_vertices
          -1)));
71    }
72    return fabricas;
73  }
74
75  int DevolverConjunto(vector<int> &set,int u){
76    int i,j;
77    int r = u;
78    while( set[r] != r ) r = set[r];
79    i = u;
80    while( i != r ){
81      j = set[i];
82      set[i] = r;
83      i = j;
84      set[i] = r;
85      i = j;
86    }
87    return r;
88  }
89
90  void UnirConjuntos(vector<int> &set,vector<int> &height,int u,int v){
91    if( height[u] == height[v] ){
92      height[u]++;
93      set[v] = u;
94    }else{
95      if( height[u] > height[v] )
96        set[v] = u;
97      else set[u] = v;
98    }
99  }
100
101 void ImprimirResultado(VectorAristas &a,int cantidad_vertices){
102   int costo = 0;
103   for( unsigned int i = 0; i < a.size();i++ ){
104     costo += a[i].first;
105   }
106   cout << costo << " ";
107   for( unsigned int i = 0; i < a.size();i++ ){
108     if( a[i].second.first < cantidad_vertices - 1 && a[i].second.second <
          cantidad_vertices -1 )
109       cout << a[i].second.first + 1 << " " << a[i].second.second + 1<< " ";
110   }
111   cout << endl;
112 }
```

Listing 7: ej3.h

```
1  typedef int Vertice;
```

```
typedef pair<int,pair<Vertice,Vertice> > Arista;
typedef vector<Arista> VectorAristas;
typedef vector< vector<int> > VectorAdyacencias;
enum ColorNodo{BLANCO,GRIS,NEGRO};

typedef struct VerticeDFS_{
  ColorNodo color;
  int distancia;
  Vertice predecesor;
} VerticeDFS;

typedef struct {
  int cantidad_vertices;
  int cantidad_aristas;
  vector<VerticeDFS> vertices;
  VectorAdyacencias adyacencias;
  VectorAristas vector_aristas;
}Grafo;

int  cargarGrafo(Grafo &g);
void ImprimirResultado(VectorAristas &a,int cantidad_vertices);
void AGMKruskal(Grafo &g,VectorAristas &a);
int DevolverConjunto(vector<int> &set,Vertice u);
void UnirConjuntos(vector<int> &set,vector<int> &height,Vertice u,Vertice v
    );
```

Listing 8: random_graph.cpp

```
int main(int argc,char **argv){
  if(argc < 4){
    cout << "Usage: " << argv[0] << " <max_n> <max_weight> <%fabricas> " <<
        endl;
    exit(1);
  }
  double max_n = atof(argv[1]);
  double pc = atof(argv[3]);
  int max_w = atoi(argv[2]);
  int aristas = 0;
  srand(time(NULL));
  int fabricas = (int)( max_n * pc );
  if( fabricas == 0 )
    fabricas++;
  int clientes = (int)max_n - fabricas;
  cout << fabricas << " " << clientes << " " << ((max_n * (max_n -1 )) /2 )
      << endl;
  for( int i = 1; i <= max_n; i++)
    for( int j = i + 1; j <= max_n;j++){
      cout << i << " " << j << " " << ( rand() % max_w ) + 1 << endl;
    }
  cout << "0" << endl;
  return 0;
}
```

5.3 Bibliografía

Referencias

[1] Brassard, Bratley, *Fundamental of Algorithmics*, Chapter 6.3, «Graphs: Minimum spanning trees», Chapter 8, «Dynamic Programming», Chapter 9.2, «Traversing trees», Chapter 9.3 «Depth-first search: Undirected graphs», Chapter 9.5 «Breadth-first», Prentice Hall, 1996

[2] Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*,Chapter 21,«Data Structures for Disjoint Sets»,Chapter 23,«Minimum Spanning Trees», The MIT Press, McGraw-Hill, 2009, Third Edition

[3] `http://www.cplusplus.com/reference/stl/`