DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico N° 2

Segundo Cuatrimestre 2017 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Nicolás Risso	883/12	ndrisso@gmail.com
Lucas Rodriguez	593/14	rodriguez.lucas.e@gmail.com
Gabriel Scotillo	527/14	gabrielscotillo@gmail.com
Pedro Facundo Tamborindeguy	627/11	pftambo@gmail.com

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

Índice

1. Ejercicio 1

1.1. Problema: Impresiones ordenadas

Se tienen que realizar n trabajos y para eso se cuenta con dos máquinas idénticas. Cada trabajo se puede realizar en alguna de las dos máquinas y tiene un costo asociado; el mismo depende únicamente del trabajo realizado justo antes en esa máquina. Además los trabajos tienen que ser realizados en orden. Es importante notar que la variable temporal no se tiene en cuenta, esto es, no interesa la duración de cada trabajo como tampoco el tiempo total que lleva realizarlos todos.

El problema que se busca resolver es encontrar alguna asignación optima de los trabajos a las máquinas tal que el costo total de realizar todos los trabajos sea mínimo. El costo total es la sumatoria del costo de realizar cada trabajo.

Para cada trabajo se conoce el costo de realizarlo. El mismo puede ser variable y depende del último trabajo realizado en la máquina. Al costo de realizar el trabajo i-ésimo dado que el último realizado en la máquina fue el j-ésimo lo vamos a notar como $C_{i|j}$. Como los trabajos se comienzan a numerar desde 1, diremos que el trabajo 0 es cuando en la máquina aún no se realizó ningun trabajo. Hay 2 trabajos iniciales, uno para cada máquina. De esta forma, $C_{7|0}$ representa el costo de realizar el trabajo 7 dado que es el primero a ser realizado en esa máquina.

Veamos un ejemplo. Supongamos que se tienen 3 trabajos: t_1, t_2 y t_3 con costos $C_{1|0}$, $C_{2|0}$, $C_{2|1}$, $C_{3|0}$, $C_{3|1}$ y $C_{3|2}$. Es importante notar que, como se tienen que realizar en orden, no existen los costos $C_{i|j}$ con $i \leq j$ porque representarían el costo de realizar el trabajo i dado que se realizó antes j en esa máquina, pero i se tenía que realizar antes que j porque $i \leq j$, por lo tanto esta situación no puede darse.
Como no importa el tiempo, podría suceder que convenga realizar todos los trabajos en una misma máquina; también podría pasar que algunos resulte mejor asignarlos a una máquina y los otros a la otra.
Lo que si tiene que suceder es que los trabajos asignados a las máquinas respeten el orden en cada una de ellas: esto es para todo par de trabajos i, j asignados a la máquina M, si $i < j$ entonces t_i se realizó antes que t_j en M.

Asignando valores al ejemplo anterior. Sean los trabajos t_1, t_2, t_3 con los siguientes costos:

$$\begin{bmatrix} C_{1|0} & & \\ C_{2|0} & C_{2|1} & \\ C_{3|0} & C_{3|1} & C_{3|2} \end{bmatrix} = \begin{bmatrix} 5 & & \\ 1 & 2 & \\ 3 & 4 & 5 \end{bmatrix}$$

Para este caso una solución óptima es asignar el trabajo 1 y 3 a una máquina y el 2 a la otra, o el 1 y 2 a una y el 3 a la otra. Ambas soluciones tienen un costo de 10. La otra posible asignación $M_1 = \{1\}$ y $M_2 = \{2, 3\}$, o $M_2 = \{1\}$ y $M_1 = \{2, 3\}$ tienen un costo de 11 por lo que no son soluciones óptimas para el ejemplo.

Veamos otro ejemplo mas. Se tienen 4 trabajos.

$$\begin{bmatrix} C_{1|0} & & & \\ C_{2|0} & C_{2|1} & & \\ C_{3|0} & C_{3|1} & C_{3|2} & \\ C_{4|0} & C_{4|1} & C_{4|2} & C_{4|3} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 2 & 2 & 1 & \\ 2 & 2 & 2 & 1 \end{bmatrix}$$

La asignación óptima para este ejemplo es $M_1 = \{1, 2, 3, 4\}$ y $M_2 = \{\}$. Es importante notar que asignar todos los trabajos a M_1 y ninguno a M_2 es la misma solución que asignar todos a M_2 y ninguno a M_1, debido a que las máquinas son idénticas y no influyen en los costos.

$$[M_1 = \{1, 2, 3, 4\}, M_2 = \{\}] \equiv [M_1 = \{\}, M_2 = \{1, 2, 3, 4\}]$$

1.2. Algoritmo

1.2.1. Idea y correctitud

Para resolver el problema se utilizó una técnica conocida como programación dinámica.
Sean dos máquinas idénticas y $t_1 \dots t_n$ los trabajos a realizarse. Definimos la siguiente función:

$f(k,i)$ ="Costo mínimo de realizar los trabajos $t_k \dots t_n$ en las máquinas, dado que los últimos trabajos realizados fueron t_{k-1} y t_i".

Se puede ver que $f(1,0)$ es la respuesta al problema porque estaríamos calculando el mínimo costo de realizar los trabajos $t_1 \dots t_n$ (todos) y los trabajos iniciales son 0 y 0, que es justamente cuando en las máquinas no se realizaron trabajos aún.

Antes de proponer una definición matemática para la función f, algunas observaciones:

- **Para cada trabajo hay únicamente dos posibilidades:**
  O bien se realiza en una máquina o bien se realiza en la otra, y esas son todas las opciones.
- **Veamos que "pinta" tiene una solución al problema:**
  Sea t_i el i-ésimo trabajo a ser realizado, por el orden que tiene que tener la solución, previo al i se tuvieron que haber asignado los trabajos $t_1 \dots t_{i-1}$. Sean t_x, t_y los últimos trabajos que se realizaron en cada una de las máquinas. ¿Puede $t_x \neq t_{i-1} \wedge t_y \neq t_{i-1}$? Supongamos que si, entonces ni t_x ni t_y son t_{i-1}; pero como se esta por realizar el trabajo i, el $i-1$ seguro ya fue asignado a alguna máquina y este no aparece como último realizado, entonces se tuvieron que haber realizado otros después del t_{i-1}; sean $t_{z_1} \dots t_{z_m}$ esos trabajos, como se están asignando los trabajos en orden y $t_i \notin \{t_{z_1} \dots t_{z_m}\} \wedge t_{i-1} \notin \{t_{z_1} \dots t_{z_m}\} \Rightarrow max\{t_{z_1} \dots t_{z_m}\} < t_{i-1} < t_i$ pero $t_{z_1} \dots t_{z_m}$ aparecen después del trabajo t_{i-1}, absurdo. **Si se esta por asignar el trabajo t_i, entonces siempre en alguna de las máquinas esta como último trabajo realizado el t_{i-1}.**

Proponemos a la función f como:

$$f(k,i) = \begin{cases} min\{C_{k|i}, C_{k|k-1}\} & \text{si } k = n \\ min\{C_{k|i} + f(k+1, k-1), C_{k|k-1} + f(k+1, i)\} & \text{otro caso} \end{cases} \tag{1}$$

Primero veamos que la recursión vale: Principio de optimalidad
Sea S una solución óptima para los trabajos $t_k \dots t_n$ dado que los últimos trabajos realizados fueron t_{k-1} y t_i. Como vimos anteriormente para el trabajo t_k hay solo dos opciones:

- $t_k \in M_1$:

$$\underbrace{M_1 = \{t_k, \underbrace{t_{a_1}, \dots, t_{a_r}}_{S'}\}, M_2 = \{t_{b_1}, \dots, t_{b_s}\}}_{S}$$

¿Puede S' no ser óptima? Sea S'' una solución para los trabajos $t_{k+1} \dots t_n$ que tiene como últimos trabajos realizados a t_k y t^* donde t^* puede ser o bien t_{k-1} o bien t_i, tal que $costo(S'') < costo(S')$.

$$costo(\{t_k\} \cup S') = C_{k|\overline{t^*}} + costo(S') \text{ , donde } \overline{t^*} = \begin{cases} i & \text{si } t^* = t_{k-1} \\ k-1 & \text{si } t^* = t_i \end{cases}$$

como es indistinta la etiqueta de las máquinas, para S'' la máquina que comienza utilizando como último trabajo realizado al t_k la llamo M_1 y a la otra M_2.

$$costo(\{t_k\} \cup S'') = C_{k|\overline{t^*}} + costo(S'') \text{ , donde } \overline{t^*} = \begin{cases} i & \text{si } t^* = t_{k-1} \\ k-1 & \text{si } t^* = t_i \end{cases}$$

$$costo(S'') < costo(S') \Leftrightarrow C_{k|\overline{t^*}} + costo(S'') < \underbrace{C_{k|\overline{t^*}} + costo(S')}_{costo(S)}$$

Conseguimos una solución para el problema $t_k \dots t_n$ que es mejor que S. Absurdo, S es una solución de costo mínimo .

- $t_k \in M_2$:
  Este caso es análogo al anterior.

La función calcula el mínimo costo:
Estamos analizando donde asignar el trabajo t_k y sabemos que los últimos realizados fueron t_{k-1} y t_i. Nuevamente hay dos posibilidades:

- **Asignamos t_k en la máquina donde el último trabajo realizado fue t_i:**

  $$\text{costo} = C_{k|i} + f(k+1, k-1)$$

  Estamos agregando t_k después de t_i, eso aumenta el costo en $C_{k|i}$. Luego hay que resolver el problema para $t_{k+1} \dots t_n$ y los últimos trabajos realizados fueron: t_k que lo acabamos de agregar y t_{k-1} que estaba de antes.

- **Asignamos t_k después de t_{k-1}**

  $$\text{costo} = C_{k|k-1} + f(k+1, i)$$

  Este caso es análogo al anterior. Ahora al agregar t_k después de t_{k-1}, los últimos trabajos realizados son t_k y el que estaba de antes: t_i.

Como a priori no sabemos cual decisión es mejor nos quedamos con la que resulte menor, entonces el costo mínimo será: $min\{C_{k|i} + f(k+1, k-1), C_{k|k-1} + f(k+1, i)\}$. Para el caso base la argumentación es la misma.

1.2.2. Pseudocódigos

Algorithm 1: dp

Data: *costos*, *n*:cantidad de trabajos
Result: *costoMin*, *trabajos*: asignados a una máquina

```
1 dicc ← CrearMatriz(tamaño : costos, inicializada : NIL);
2 min ← dpRecur(1, 0);
3 trabajos ← construirSol(costos, dicc);
4 return min, trabajos
```

Algorithm 2: dpRecur

Data: k, i, variables globales: *costos*, n, *dicc*
Result: *costoMin*

```
1 if no esta definido dicc[k][i] then
2 |   if k = n then
3 |   |   dicc[k][i] ← min{costos[k][i], costos[k][k − 1]}
4 |   else
5 |   |   dicc[k][i] ← min{costos[k][i] + dpRecur(k + 1, k − 1), costos[k][k − 1] + dpRecur(k + 1, i)}
6 |   end
7 end
8 return dicc[k][i]
```

Algorithm 3: construirSol

Data: *costos*, *optimos*
Result: *trabajos*: asignados a una máquina

```
1  M1, M2 ← CrearVector(con el elemento 0);
2  M1 + [1];
3  ultimo ← 0;
4  for i = 2, ..., n-1 do
5  |  c ← costos[i][M1.ultimoElemento()];
6  |  if optimos[i][ultimo] - c = optimos[i+1][M2.ultimoElemento()] then
7  |  |  M1 + [i];
8  |  |  ultimo ← M2.ultimoElemento();
9  |  else
10 |  |  M2 + [i];
11 |  |  ultimo ← M1.ultimoElemento();
12 |  end
13 end
14 c ← costos[i][M1.ultimoElemento()];
15 if optimos[i][ultimo] - c = 0 then
16 |  M1 + [i];
17 |  ultimo ← M2.ultimoElemento();
18 else
19 |  M2 + [i];
20 |  ultimo ← M1.ultimoElemento();
21 end
22 M1.eliminarPrimerElemento();
23 M2.eliminarPrimerElemento();
24 return M1
```

El pseudocódigo de la función $dpRecur(k, i)$ respeta exactamente la definición matemática de la función $f(k, i)$ con el agregado de un diccionario para no calcular un sub-problema más de una vez. El mismo esta implementado sobre una matriz de $n \times n$. Se tiene acceso en tiempo constante, por lo que todas las operaciones sobre el diccionario son $O(1)$ excepto cuando se lo crea y se inicializa que cuesta $O(n^2)$. La función $dp()$ calcula la solución al problema, primero crea el diccionario vacío, luego llama a la función $dpRecur(1, 0)$ para obtener el mínimo, por último llama a la función $construirSol()$ que devuelve una asignación de trabajos mínima para una máquina.

Construcción de la solución:
Además de obtener el mínimo costo de asignar los trabajos a las máquinas, se quiere saber explicitamente alguna asignación de trabajos para una de las máquinas tal que alcance ese mínimo. Notar que, conociendo la asignación de los trabajos para una de las máquinas también se conocen los que fueron a la otra.
Por una cuestión de simplicidad decidimos devolver los trabajos que están en la máquina M_1. La máquina M_1 va a ser la que tenga como primer trabajo al t_1.
Los parámetros que recibe el algoritmo son la matriz con los costos y el diccionario completado por la función $dpRecur()$ luego de ser llamada con los parámetros $(1, 0)$. Notar que cada combinación k, i de claves para el diccionario, en su definición contiene a $f(k, i)$ que es el costo mínimo para los trabajos $t_k \dots t_n$ dado que los últimos realizados fueron t_i y t_{k-1}.
El algoritmo al principio comienza inicializando los trabajos de ambas máquinas, y como estas no realizaron ninguno aún, se le agregan los iniciales: el 0. Luego, al algoritmo lo podemos dividir en 3 partes:

- **El primer trabajo:** t_1
  Como t_1 siempre va primero en alguna máquina, lo agregamos a M_1 por lo mencionado anteriormente. Luego guardamos en la variable *ultimo*, el último trabajo realizado en la otra máquina donde no agregué al trabajo.

- **Trabajos desde** t_2 **hasta** t_{n-1}

Para el trabajo t_i hay dos posibilidades:

- Se agregó a M_1:
  La condición para agregar al trabajo t_i a la máquina 1 es que el costo de agregarlo a M_1 sumado al óptimo para $i+1 \dots n$ dado que los últimos trabajos realizados fueron i (que es el que quiero agregar) y el último que esta en M_2, sea igual al óptimo para los trabajos $t_i \dots t_n$ dado que los últimos fueron $i-1$ y el valor que esta en la variable *ultimo*. Pasando la oración a variables es que:

  $$costos[i][M1.ultimoElemento()] + optimos[i+1][M2.ultimoElemento()] = optimos[i][ultimo]$$

  Si la comparación resulta cierta, entonces se puede alcanzar el mínimo agregando t_i a M_1. Se lo agrega y luego se actualiza la variable *ultimo*.
- Se agregó a M_2:
  Si no se pudo agregar a M_1 entonces necesariamente tuvo que haber sido agregado a M_2. Lo agregamos y actualizamos la variable *ultimo*.

- **El último trabajo:** t_n
  Este caso es idéntico al anterior pero con la excepción que ya no quedan más trabajos, entonces tenemos que evaluar la siguiente ecuación:

  $$costos[i][M1.ultimoElemento()] = optimos[i][ultimo]$$

Por último se eliminan los trabajos iniciales 0 de las máquinas y se retorna M_1. De esta forma tenemos una asignación posible que cuesta lo mínimo.

1.2.3. Complejidad

El costo total del algoritmo $dp()$ lo podemos calcular como $C_1 + C_2 + C_3$.

- C_1 : costo de crear e inicializar la matriz.
  Crear e inicializar una matriz de $n \times n$ cuesta $O(n^2)$

- C_2 : costo de la función recursiva *dpRecur* al ser llamada con parámetros 1 y 0.
  Como se esta utilizando un diccionario para no calcular los subproblemas más de una vez y devolver el resultado en tiempo constante (asumimos tener el valor del llamado recursivo en $O(1)$), el costo total de la función lo podemos acotar por la siguiente sumatoria:

  $$\sum_{s \in subproblemas} costo(s)$$

  Y como todos los subproblemas cuestan tiempo constante porque solamente se realizan comparaciones, asignaciones y accesos a una matriz, la complejidad del algoritmo resulta:

  $$O(\#subproblemas)$$

  Resta ver cuantos subproblemas hay. La variable k comienza en 1 y aumenta hasta n cuando llega al caso base. Luego por cada uno de los valores que toma k, el i puede variar entre 0 y $k-1$. La cantidad de combinaciones de este par de valores puede ser acotada por $n^2 \Rightarrow O(n^2)$.

- C_3 : costo de la función que arma una solución óptima.
  La función realiza un ciclo lineal en n donde todas sus operaciones internas son $O(1)$ porque son comparaciones, asignaciones, inserciones a un vector de C++ [1], consultar el último y primer elemento de un vector de C++ [2]. El costo del ciclo es $O(n)$. Al final del algoritmo se eliminan[3] los primeros valores de cada vector, eso cuesta $2 \times O(n) \in O(n)$. El algoritmo en peor caso realiza $O(n)$ operaciones.

$$C_1 + C_2 + C_3 = O(n^2) + O(n^2) + O(n) \in O(n^2)$$

La complejidad total de la función es $O(n^2)$.

[1] http://www.cplusplus.com/reference/vector/vector/
[2] http://www.cplusplus.com/reference/vector/vector/back/
[3] http://www.cplusplus.com/reference/vector/vector/erase/

1.3. Experimentación

Para ambos experimentos se utilizaron entradas aleatorias de distintos tamaños. Se los hizo variar desde 5 hasta 480 aumentando de a 20. Para cada tamaño se generaron 25 instancias aleatorias con una distribución uniforme. Como el formato de los costos es siempre el mismo, lo que es aleatorio es cada costo[i][j]. Además para cada entrada se realizaron 25 mediciones de tiempo y luego se los promediaron.

1.3.1. Comparación entre dos implementaciones

Los algoritmos recursivos tienen cierto *overhead* de tiempo. El algoritmo propuesto en la sección 1.2 es un algoritmo recursivo que va calculando los valores de los sub-problemas y los guarda en un diccionario por si los necesita nuevamente poder devolver el valor rápido. Cada clave del diccionario representa los parámetros del sub-problema y su definición es la respuesta a ese sub-problema.

La respuesta al problema original es el valor de la clave $(1, 0)$; como al principio el diccionario comienza vacío, el valor para $(1, 0)$ no esta calculado aún, entonces el algoritmo realiza un llamado recursivo para calcularlo, necesita el valor de $(2, 0)$, que a su vez este necesita los valores de $(3, 0)$ y $(3, 1)$ y así sucesivamente. Si en el medio el algoritmo necesita un valor que ya calculo lo devuelve en $O(1)$ gracias al diccionario.

Podemos ver que, si de entrada ya tenemos calculado el valor para $(1, 0)$ el algoritmo no necesita realizar llamados recursivos. Ahora si vemos al diccionario como la matriz que lo implementa, llenándola de forma adecuada podemos prescindir de los llamados recursivos, transformando el algoritmo en uno iterativo y así tener menor *overhead* de tiempo. Es importante notar que para calcular cada valor (k, i) estos necesitan únicamente los valores $(k + 1, i)$ y $(k + 1, k - 1)$; es decir de la fila de abajo $(k + 1)$ y las columnas i y $k - 1$. Entonces podemos ir completando la matriz de abajo hacia arriba y de izquierda a derecha y estas dependencias se van a respetar.

Para esto transformamos el algoritmo recursivo con memoización en uno iterativo que completa la matriz y luego devuelve la posición $(1, 0)$. Al contrario que el recursivo que comienza calculando el sub-problema más grande (la respuesta al problema), el iterativo comienza con los sub-problemas más chicos y luego calcula los más grandes; esta forma de recorrer los sub-problemas se la conoce como *bottom-up* y a la recursiva como *top-down*.

En el siguiente experimento queremos comparar los tiempos de ejecución para ambos algoritmos. Si bien pertenecen a la misma clase de complejidad, suponemos que el algoritmo iterativo resultará más rápido debido a que tiene menos *overhead*, no realiza llamados recursivos.

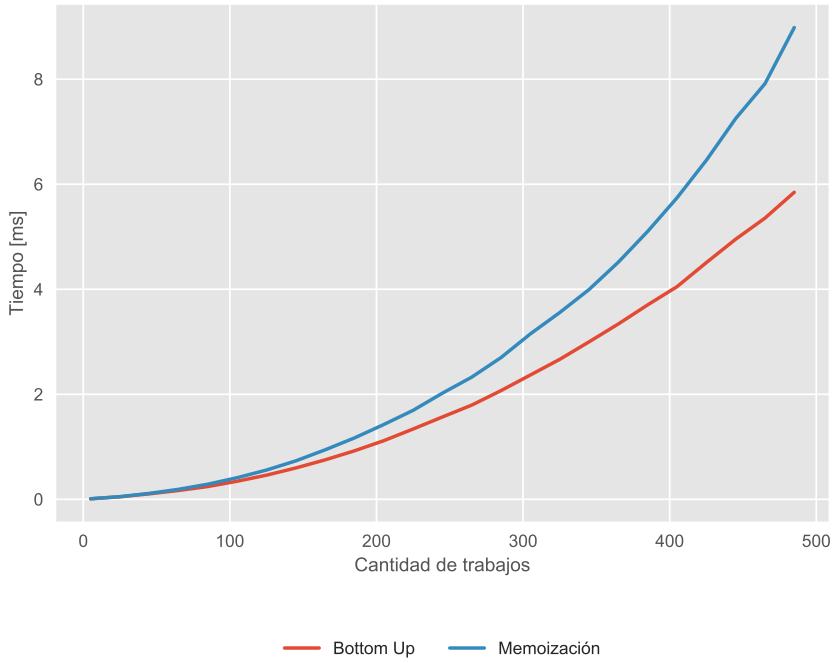


Figura 1: Comparación de tiempos de ejecución entre el algoritmo bottom-up y el recursivo con memoización: top-down.

Como era de esperarse, el algoritmo iterativo es notablemente más rápido. Lo llamativo es que la diferencia a medida que la entrada crece pareciera ser cada vez mayor.

1.3.2. Complejidad

Para analizar si la complejidad teórica se condice con la empírica, decidimos comparar los tiempos medidos contra distintas funciones utilizando el coeficiente de correlación de Pearson. Nuestra hipótesis es que, al tener una complejidad en peor caso de $O(n^2)$ para ambas implementaciones, el coeficiente sea más cercano a 1 cuando se comparan los tiempos contra la función n^2 que contra las otras.

f(n)	n	$nlogn$	n^2	n^2logn	n^3	n^4
top-down	0.949652	0.963788	0.996823	**0.998504**	0.995087	0.977116
bottom-up	0.967238	0.978952	**0.999912**	0.999447	0.986449	0.959457

Cuadro 1: Coeficientes de Pearson

En el cuadro 1 se puede observar que para la implementación *bottom-up* la correlación más fuerte se da para n^2, hay suficiente evidencia para pensar que el tiempo de ejecución es proporcional a n^2 ya que la correlación dio muy cercana a 1. En cambio para la implementación *top-down* la correlación más alta no dio para la función n^2 sino para 2^i con i entre 2 y 3, pero más cercano a 2 ya que la correlación con n^2 es mayor que para n^3. No sabemos con seguridad a que se debe esto.

2. Ejercicio 2

2.1. Problema: Replicación de contenido

Este problema trata de una organización que necesita encontrar una solución para replicar (o sincronizar) contenidos en una red de internet. Se cuenta con n servidores interconectados mediante m enlaces de alta velocidad. Se debe seleccionar uno de los n servidores como *master*, a partir del cual se iniciará la replicación de los nuevos contenidos. Este transmitirá su copia a algunos servidores, y estos inmediatamente guardarán una copia y transmitirán inmediatamente a otros servidores. Este proceso se repite hasta que todos los servidores tengan la información. Cada enlace que conecta un servidor con otro tiene un costo asociado en función del tráfico que trasnmiten, y en este caso todos lo harán con la misma información (son copias). La organización necesita minimizar dos cosas:

- 1) Los costos por utilizar los enlaces.
- 2) El tiempo de replicación.

2.1.1. Modelado mediante grafos

Modelaremos este problema mediante grafos (sin dirección). Cada *servidor* representará un **vértice** del grafo, los *enlaces* que conectan a cada uno serán las **aristas** del mismo y el **peso** de estas será el *costo* asociado por utilizar el enlace. Es importante notar dos cosas:

- Cada servidor podría estar conectado a varios servidores (por lo menos a uno, no hay servidores aislados) aunque no necesariamente con todos directamente.

- Si un servidor estuviera conectado a otro, lo haría por un único y exclusivo enlace: es decir, si el servidor x se conecta con el y, solo habrá un enlace entre ellos y tendrá un peso (costo) asociado. En caso de haber más de un enlace entre dos servidores, solo se utilizaría el de menor costo asociado.

- Si un servidor no estuviera directamente conectado con el *master* debería esperar que la información le llegue a alguno de los servidores que sí está conectado y recibir la información a través de un enlace.

Es decir, en términos de grafos estamos frente a uno grafo conexo, sin direcciones y con pesos en las aristas.

Nuestro grafo será $G = (V, X)$, con V = conjunto de vértices, X = conjunto de aristas, $n = |V|$, $m = |X|$.

2.1.2. Solución al problema 1: Minimizar costos de los enlaces

La información llegará a un servidor a través de un enlace (a excepción del *master* que será el que inicie el proceso), y lo hará por medio de un único enlace (ya que no tendría sentido pagar el costo por usar más enlaces si ya se está recibiendo el contenido por uno de ellos). La empresa quiere replicar la información en **todos** los servidores minimizando el costo de los enlaces, es decir, que la suma de los costos de los enlaces sea mínima y que la información llegue a todos los servidores.

En términos de grafos, una solución posible es encontrar el árbol generador mínimo del grafo que modela este problema. El árbol generador mínimo cumple que conecta a *todos* los vértices y lo hace a través de un subconjunto de aristas tal que la suma de ellas es *mínima* frente a cualquier otro árbol generador. Cada arista del AGM representará el enlace que deberá utilizar la organización para transmitir la información. Además, éste es conexo, por lo que la información le llegará a todo servidor. De esta manera, al conseguir el AGM del grafo estaremos encontrando los enlaces que minimizarán los costos a la empresa en la replicación de contenidos. Veamos un ejemplo:

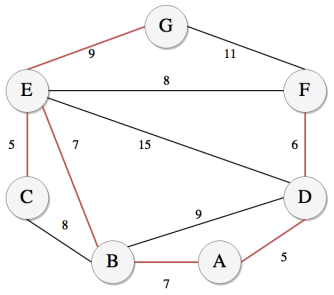


Figura 2: Las aristas en rojo forman parte de un AGM.

Para resolver esto implementamos el algoritmo de Prim [4], el cual es un algoritmo conocido para calcular el *árbol generador mínimo de un grafo*. Se utilizó la versión básica sin cola de prioridad.

Complejidad: $\mathcal{O}(n^2)$.

2.1.3. Solución al problema 2: Minimizar el tiempo de replicación

La organización quiere reducir el tiempo de replicación. El tiempo que requiere en atravesar un enlace se considera igual para todos los enlaces y el tiempo que tarda un servidor cuando recibe la información, hace su copia local y la retransmite se considera instantánea a todos sus vecinos. El inicio del proceso se hace desde el *master*. En este contexto, estamos buscando que servidor seleccionar como *master* de manera tal que la replicación termine lo antes posible. Como en todos los enlaces consideramos que cuestan la misma unidad de tiempo, vamos a tomar que el costo temporal de atravesar un enlace como 1 unidad de tiempo. De esta manera, por ejemplo, si tenemos un servidor s que se conecta a un vecino del *master* y no directamente con él, el costo temporal hacia ese servidor s será de 2 unidades de tiempo.

En términos de grafos, producto de la solución del problema 1 obtuvimos un árbol generador del grafo. Entonces, una solución posible a este problema es encontrar una raíz para el árbol de forma tal que la altura del mismo sea mínima frente a todas las posibles raices. En el ejemplo anterior:

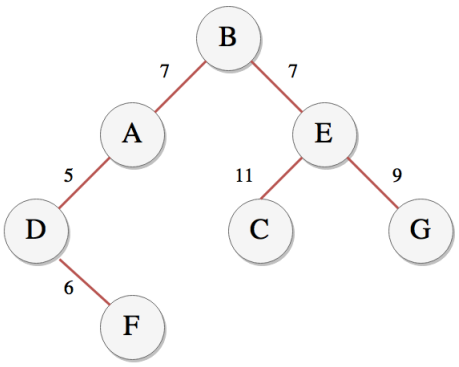


Figura 3: El nodo B es el cantidato a ser el *master*.

Para encontrar la raíz buscada, buscamos primero la rama más larga del árbol, es decir, el camino (simple) más largo que se encuentre en el mismo. Una vez obtenido este camino, nos quedamos con el vértice del medio (o alguno de ellos, en caso de tener una cantidad par) y lo seleccionamos como *master*.

- Encontrar la rama más larga:

  - Desde cualquier nodo v, ejecutamos BFS sobre el árbol, obteniendo el nodo u más lejano a v. Esto es así, ya que BFS recorre el árbol en anchura, por lo que finalizará de recorrerlo en la distancia más lejana a v, el cual será una punta de la rama más larga del árbol. Luego de obtener v, se lanza nuevamente BFS desde él, obteniendo w la otra punta de la rama más larga. Luego sabemos que la rama más larga está formada por el camino entre v y w. Obtenemos los nodos entre ellos con el algoritmo DFS.

- Seleccionamos el elemento del medio (o uno de los dos en caso de que sea una cantidad par de elementos en el camino) como la raíz del árbol. De esta manera minimizamos la distancia a cada nodo. Para hacerlo, ejecutamos DFS desde uno de los vértices obtenidos hasta llegar al otro y guardarnos los vértices a lo largo del mismo.

2.1.3.1 Correctitud

Primero notemos que estamos frente a un grafo que es árbol el cual fue generado con el algoritmo de Prim, y estos cumplen que hay un único camino simple entre todo par de nodos. Es decir, la distancia entre u y v va a estar dada por la longitud del único camino simple entre ellos. Los notaremos $\mathcal{D}(u, v) = \mathcal{L}(P_{uv})$.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 23.2: The algorithms of Kruskal and Prim, pp.567–574.

Rama más larga:

Sean s y t los extremos de la rama más larga del árbol, P_{st}. Si lanzamos BFS desde cualquier nodo u, llegamos a un nodo v que necesariamente va a ser una hoja del árbol pues sino podríamos seguir expandiéndonos. Separémoslo en caso:

Si $v \in P_{st}$, v tiene que ser s ó t. Supongamos que no fuera así, $\mathcal{D}(u,v) > max(\mathcal{D}(u,s), \mathcal{D}(u,t))$. pero de esta manera podríamos armar un camino más largo entre v y s ó t, lo que contradice el hecho de que P_{st} sea la rama más larga. Por lo tanto v es s ó t, y nuevamente lanzando BFS desde v obtendremos el otro extremo de la rama.

Si $v \notin P_{st}$:

- a) Sea P_{uv}, el camino más largo obtenido por BFS desde u a v. Supongamos que P_{uv} y P_{st} comparten por lo menos un vértice, llamémos y a la primer intersección. Por lo anterior, como $y \in P_{st}$, caemos en el anterior caso, BFS debe terminar en s ó t.

- b) Supongamos que Puv y Pst no comparten ningún vértice. Como el árbol es conexo, debe existir un $z \in Puv$ tal que tenga camino hacia un $y \in Pst$. Como P_{uv} es el camino más largo empezando desde u, $\mathcal{D}(u,z) + \mathcal{D}(z,y) + \mathcal{D}(y,t) < \mathcal{D}(u,v)$. De manera similar obtenemos, $\mathcal{D}(v,z) + \mathcal{D}(z,y) + \mathcal{D}(y,s) < \mathcal{D}(u,v)$.

Sumando ambas expresiones obtenemos,

$$\mathcal{D}(u,v) + 2 * \mathcal{D}(z,y) + \mathcal{D}(s,t) < 2 * \mathcal{D}(u,v)$$
$$2 * \mathcal{D}(z,y) + \mathcal{D}(s,t) < \mathcal{D}(u,v)$$
Y finalmente tenemos que,
$$\mathcal{D}(s,t) < 2 * \mathcal{D}(z,y) + \mathcal{D}(s,t) < \mathcal{D}(u,v)$$

Lo cual es absurdo pues P_{st} era la rama más larga del árbol. Nuevamente, como v es s ó t lanzando BFS obtendremos el otro extremo.

Vértice del medio como raíz minimiza la altura del árbol: Tenemos entonces la rama más larga del árbol, P_{st} tal que $\mathcal{D}(s,t)$ es máxima. Veamos que si tomamos un vértice en este camino como raíz, la altura del árbol va a ser por lo menos $\lceil \mathcal{D}(s,t)/2 \rceil$ y si elegimos la raíz fuera de este camino, la altura va a ser mayor que $\lceil \mathcal{D}(s,t)/2 \rceil$.

Elijamos entonces la raíz r como el vértice del medio en el camino P_{st}, de manera que $\mathcal{D}(s,r) = \lceil \mathcal{D}(s,t)/2 \rceil$ y $\mathcal{D}(r,t) \leq \lceil \mathcal{D}(s,t)/2 \rceil$. Supongamos que hay otro vértice w tal que $\mathcal{D}(r,w) > \lceil \mathcal{D}(s,t)/2 \rceil$ y por lo tanto $\mathcal{D}(r,w) > \mathcal{D}(s,r) \geq \mathcal{D}(r,t)$. Como en un árbol existe un único camino simple entre todo par de nodos, tenemos que $\mathcal{D}(s,t) = \mathcal{D}(s,r) + \mathcal{D}(r,t) < \mathcal{D}(s,r) + \mathcal{D}(r,w) = \mathcal{D}(s,w)$, por lo tanto $\mathcal{D}(s,t) < \mathcal{D}(s,w)$ lo cual es absurdo pues contradice que P_{st} sea el camino más largo.

2.1.3.2 Complejidad

Sabemos que al estar frente a un árbol, la cantidad de aristas es $m = n - 1$, por lo tanto $m \in \mathcal{O}(n)$. En este algoritmo lo que se hace es:

- Ejecutar dos veces BFS para encontrar los nodos de la rama más larga, $\mathcal{O}(n+m) = \mathcal{O}(n)$
- Ejecutar una vez DFS para reconstruir el camino entre los nodos obtenidos, $\mathcal{O}(n+m) = \mathcal{O}(n)$.
- Recorrer la lista de nodos que representa el camino hasta la mitad y obtener el del medio, $\mathcal{O}(n)$

Complejidad total: $\mathcal{O}(n)$

2.1.4. Experimentación

En esta sección presentamos una serie de experimentos en base a la implementación de la solución propuesta.

Los algoritmos se ejecutarán en el entorno: Lenovo Thinkpad T430, procesador Intel Core I5-3210M @ 2.5Ghz x 4, 12 GB de memoria RAM, sistema operativo Ubuntu 14.04 LTS (64 bits).

2.1.4.1 Experimento 1: Complejidad

En este experimento analizaremos la complejidad temporal del algoritmo. En la sección anterior vimos que la complejidad era $\mathcal{O}(n^2)$. Se corrió el algoritmo frente a tres tipos de grafos para los cuales se fue variando la cantidad n de vértices desde $n = 10, 11, 12, ..., 500$:

- El primer tipo, se trata de grafos completos $\mathcal{K}_n$, los cuales se caracterizan por tener todas las aristas posibles.
- El segundo tipo, se trata de grafos esparsos, los cuales se caracterizan por tener una densidad baja. (en nuestro caso, entre 0.045 y 0.26).
- El tercer tipo, se trata con grafos con cantidad de aristas random (densidad entre 0.10 y 1). Es decir, para un n dado, se corrio el algoritmo con 20 cantidades de aristas m_1,...,m_{20} distintas. Luego, se tomo la mediana de los ticks de reloj que arrojaron (la densidad varió uniformemente entre 0.045 y 1) .

Nota: Para cada set de inputs, es decir dados n y m fijos, se corrió el input varias veces y se tomó la mediana y los pesos de las aristas fueron random entre 1 y 100.

Los resultados fueron:

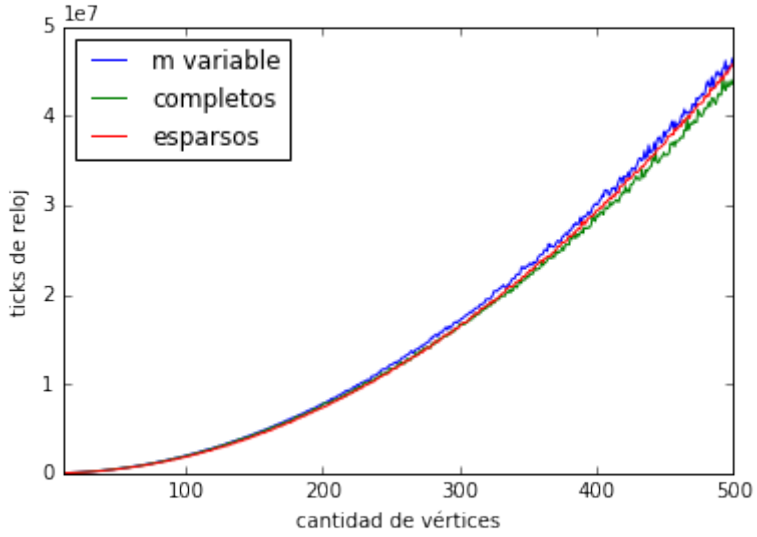


Figura 4: Primer análisis de complejidad

Podemos ver en la figura 1 que el algoritmo se comporta parecido frente a los distintos tipos de grafos, es decir, no parece haber mucha diferencia entre los grafos completos, los grafos con una cantidad variable de aristas o los esparsos. Por otro lado, podemos ver que en los tres casos que a medida que el n crece, la curva parece crecer supralinealmente.

En un segundo análisis, decidimos analizar los datos frente al coeficiente de correlación de Pearson frente a distintas funciones polinomiales, esperando que se vea reflejada una correlación positiva frente a la función cuadrática. A continuación presentamos los datos en la siguiente tabla:

$f(n)$	n	$n^{1,75}$	n^2	n^3	n^4	$nlogn$
completos	0.973508	0.999149	**0.999792**	0.984017	0.955397	0.818443
esparsos	0.970620	0.998676	**0.999977**	0.986186	0.958785	0.812015
m variable	0.972783	0.999054	**0.999843**	0.984419	0.955908	0.816559

Cuadro 2: Coeficientes de Pearson

Como podemos observar en el cuadro 1, en los tres casos distintos de grafos, la mayor correlación se da con la función $f(n) = n^2$ (coeficiente más cercano a 1), la cual pertecene al conjunto de funciones que caracteriza la complejidad de nuestro algoritmo: $f(n) \in \mathcal{O}(n)$.

2.1.4.2 Experimento 2: Prim con arreglo vs Prim con cola de prioridad

Para el siguiente experimento, implementamos una versión alternativa al algoritmo de Prim. Esta se basa principalmente en usar una cola de prioridad (heap) para almacenar las distancias, en vez del arreglo que se utiliza en la versión básica. Esto hace que la búsqueda (y extración) del vértice con menor distancia en cada iteración se realize con un costo temporal $\mathcal{O}(log(n))$ frente al costo lineal de la versión básica. Quedando las complejidades temporales totales de la siguiente manera:

- Implementación básica de Prim, complejidad total: $\mathcal{O}(n^2)$
- Implementación con cola de prioridad de Prim, complejidad total: $\mathcal{O}(m + nlog(n))$

Notemos que la complejidad implementación de Prim con cola de prioridad depende también de la variable m. Si bien solo nos fijáramos en la variable n podríamos deducir que ésta última versión es mejor ya que $nlog(n)$ es mejor que n^2. Sin embargo, esta conclusión m podría variar mucho para cada tipo de grafo. Por ejemplo, en el caso de grafos completos $m = \frac{n*(n-1)}{2}$, por lo tanto $m \in \mathcal{O}(n^2)$ y para el caso de grafos árboles, $m = n - 1$, por lo tanto $m \in \mathcal{O}(n)$.

Frente a este primer análisis, nuestra hipótesis es que frente a grafos completos (o muy densos) el algoritmo que mejor va a funcionar va a ser el de la implementación básica (con arreglo) y frente a grafos esparsos (de poca densidad) el que mejor funcionará será el que implementa con cola de prioridad. Frente al tercer caso, el de los grafos que dado un n fijo variamos las m aristas (con una densidad random entre 0.10 y 1) a priori no tenemos muchas certezas de cual será el comportamiento, por lo que será un caso experimental sin una hipótesis sólida.

Corrimos ambos algoritmos frente a tres distintos tipos de grafos, construídos de igual manera que en el experimento 1. Los primeros resultados fueron:

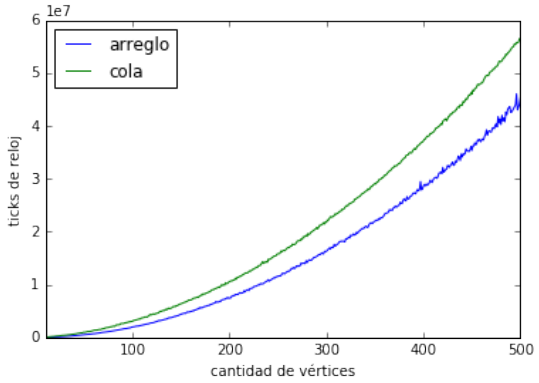


Figura 5: Grafos completos.

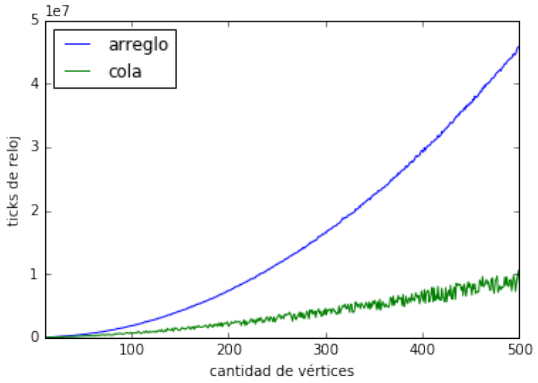


Figura 6: Grafos esparsos.

Podemos observar de ambas figuras, que como suponíamos, el algoritmo con arreglo funciona mejor frente a grafos completos, mientras que el de la cola de prioridad funciona mejor frente a grafos esparsos. Aunque con un análisis más fino también podemos observar que la relación entre ambos grafos es distinta, es decir, si bien el algoritmo con arreglo funciona mejor frente a grafos completos, en el caso de los grafos esparsos la diferencias es notablemente mayor entre ambos algoritmos. Frente a este detalle, podemos suponer adicionalmente al experimento que, podríamos haber generado grafos con mayor densidad (esta varió entre 0.15 y 0.26) y el algoritmo con cola de prioridad se seguiría comportando mejor. Veamos ahora que sucedió con los grafos con cantidad de aristas variables:

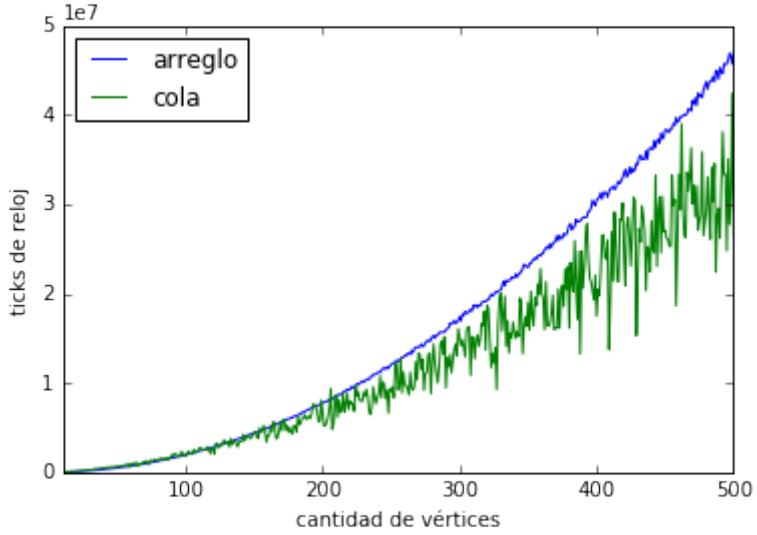


Figura 7: Grafos con m variable (random)

De este última figura, podemos ver que en casi todos los casos (para no decir todos) el algoritmo con cola de prioridad funcionó mejor que el algoritmo con arreglo, recordemos que estamos desde grafos muy esparsos hasta muy densos (cercanos a completos). No está de más notar que es esperable que la curva del algoritmo con cola de prioridad fluctúe tanto, ya que su complejidad depende del m y estamos variando mucho ésta cantidad.

Motivados por el análisis anterior y este último resultado, se procedió a realizar una segunda parte de este experimento. Este consistió en fijar la cantidad de vértices n y variar la cantidad de aristas desde el mínimo ($m = n - 1$) hasta el máximo ($m = \frac{n*(n-1)}{2}$). En este caso, la densidad d variará entre 0.05 y 1. Se realizó el experimento para $n \in \{50, 100, 200, 500\}$. Veamos los resultados:

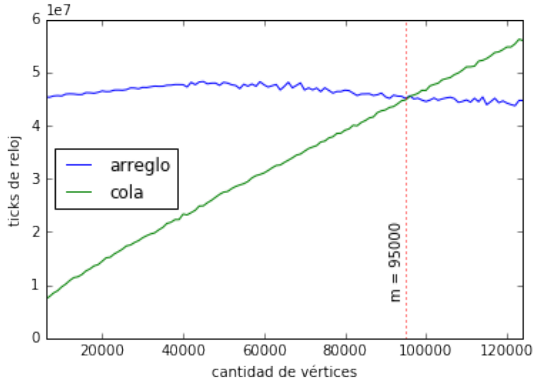


Figura 8: $N = 500$.

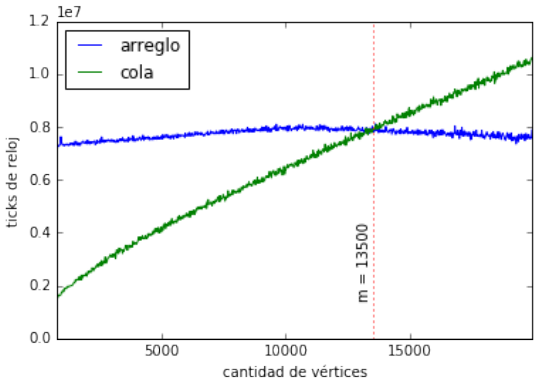


Figura 9: $N = 200$.

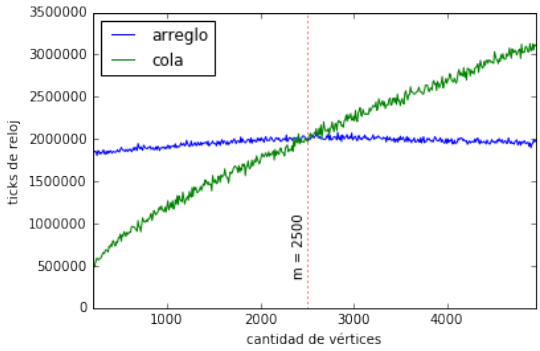


Figura 10: $N = 100$.

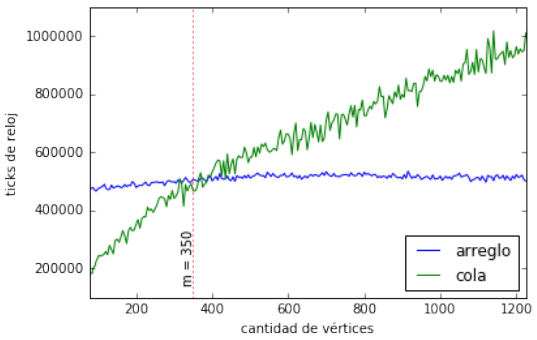


Figura 11: $N = 50$.

Podemos ver en todas las figuras que al principio (grafos esparsos) el algoritmo con cola de prioridad es

mejor y que a partir de una cierta cantidad de aristas m las curvas de los algoritmos se curvan y pasa a ser el algoritmo con arreglo el mejor. También podemos observar que a medida que para n mayor, el punto de cruce de las curvas sucede más "tarde"(las líneas rojas punteadas arrancan con $n = 500$ en el sector derecho de la figura y con $n = 50$ lo hace cerca del sector izquierdo). Si analizamos en el punto de cruce de las curvas como d, la densidad que alcanzan los grafos en ese momento, podemos ver que a medida que n crece, d también lo hace. Veamos el siguiente cuadro:

n	50	100	200	500
densidad	0.2857	0.5050	0.6783	0.7615

Cuadro 3: Densidades en los puntos de cruce

3. Ejercicio 3

3.1. Problema: Transportes pesados

Este problema trata sobre una empresa transportista de ladrillos que para poder abastecer a sus clientes tiene que fortalecer las rutas hacia ellos. La empresa quiere minimizar sus gastos, por lo que deberá elegir ciertas rutas tales que su costo de fortalecimiento sea mínimo y a su vez le permita llegar a todos sus clientes.

La empresa cuenta con una cantidad F de fábricas y tiene C clientes que abastecer. Los clientes pueden ser abastecidos desde cualquiera de las fábricas. En total hay R rutas que conectan a las fábricas y a los clientes. Cada ruta puede conectar a dos fábricas, a dos clientes o a un cliente con una fábrica. Sabemos además que todas las fábricas tienen al menos una ruta que llega a ella y que la demanda de los clientes puede ser satisfecha, esto es que para todo cliente existe un camino (directo o no) que llega hacia alguna fábrica. Cada ruta tiene asociado un costo de fortalecimiento que es proporcional a su longitud. Por último, también se sabe que no hay más fábricas que clientes.

El problema deberá resolverse con una complejidad temporal en peor caso de $\mathcal{O}(C^2)$ o bien $\mathcal{O}(RlogC)$.

Este problema será resuelto mediante un grafo con pesos en las aristas en el que en un principio los vértices serán los clientes y fábricas, y las aristas serán las rutas existentes entre clientes, fábricas como también entre cliente - fábrica.

Sabemos que la necesidad de todo cliente es satisfacible mediante rutas hacia una fábrica por lo que existe un *camino* entre estas, y como se quiere minimizar el gasto, debemos elegir de manera óptima rutas -es decir aristas- tal que la inversión total sea mínima, preservando para todo cliente lo dicho anteriormente.

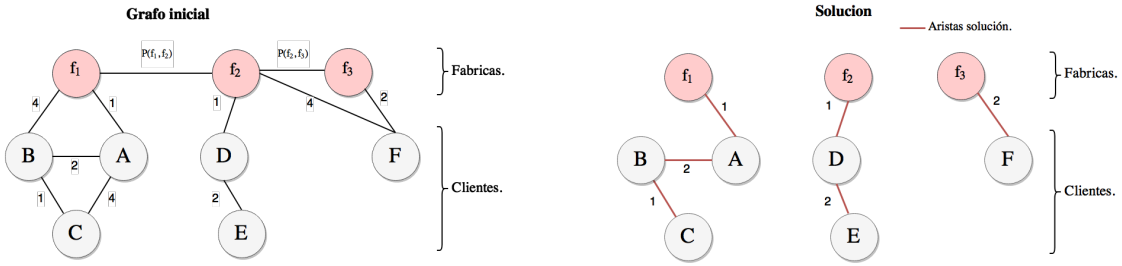
Una solución al problema es una lista de aristas tal que su suma es mínima y además permite satisfacer la demanda de los clientes.

Observación: ¿Puede una solución contener una arista entre dos fábricas?
Supongamos que si.
Sea $e = (F_1, F_2)$ la arista que conecta a dos fábricas y $e \in S$ solución.
Como es solución, todo cliente es abastecido.
Si uno de ellos es abastecido por F_1 utilizando a e, la demanda será satisfacicle desde F_2.
Si uno de ellos es abastecido por F_2 utilizando a e, la demanda será satisfacicle desde F_1.
Removiendo a e de la solución, seguimos cumpliendo con la demanda de los clientes.
Luego, $S - \{e\}$ es solución y es mejor que S pues $peso(e) > 0$.
Absurdo! Pues S era mínima.

Observación: ¿Puede una solución contener un ciclo simple?
Supongamos que si.
Sea c el ciclo formado por las aristas $e_1, e_2, \ldots, e_n \in S$ solución.
Removemos $e_i \in c$.
Por lo visto en la observación anterior, e_i en sus extremos no tenía fábricas porque S es solución.
Entonces, o bien conectaba a dos clientes o bien a un cliente con una fábrica.
Si ocurre lo primero y además, alguno de ellos utilizaba e_i para llegar a una fábrica, el cliente puede llegar a esta última utilizando la otra parte de c.
Sino, si ocurre lo segundo, la demanda del mismo es satisfacible utilizando la otra parte de c.
Removiendo a e_i de la solución, seguimos cumpliendo con el problema.
Luego, $S - \{e_i\}$ es solución y es mejor que S pues $peso(e_i) > 0$.
Absurdo! Pues S era mínima.

Podemos concluir que una solución al problema no deberá contener ciclos simples ni aristas entre fábricas. La solución es un bosque, grafo que no tiene ciclos.

Veamos un ejemplo con su solución. En este caso hay una única solución pero esto no tiene porque pasar siempre.



(a) Grafo original.

(b) Grafo solución.

Figura 12: Ejemplo con su solución.

3.2. Algoritmo

Veamos la idea del algoritmo que resuelve el problema. Si en el grafo inicial hubiese una única fábrica, por las características del problema, este sería *conexo* dado que para todo cliente existe al menos una fábrica a la cual puede llegar, en particular, llega a la única fabrica que hay. Un árbol generador mínimo de ese grafo nos provee la mínima cantidad de aristas tal que, es *conexo* (pues árbol), contiene a todos los nodos (por ser generador) y además, la suma de los pesos de las aristas es mínima. Entonces tendríamos la mínima cantidad y peso de aristas talque que el grafo es conexo. Asegurándonos que para cada cliente existe un camino hacia la única fábrica. Calcular el AGM cuando el grafo inicial es conexo y hay una sola fábrica y luego quedarnos con las aristas nos da la respuesta al problema.

El problema original no tiene porque tener un grafo inicial con una sola fábrica, por esta razón no podemos calcularle el AGM directamente y así obtener la solución.

Para salvar esta situación podemos agregar un nodo ficticio y unirlo a todas las fábricas con ejes de costo 0. Recordar que las rutas originalmente tienen un costo positivo. Hacer esto es equivalente a tener una única fábrica porque partiendo desde cualquier nodo al llegar a una fábrica luego podemos usar el nodo ficticio y movernos hacia otra con costo 0 independientemente si existía un eje entre esas fábricas; entonces una vez que se llega a una fabrica sin costo adicional se puede llegar a otra, podemos pensar como que el nodo ficticio es la única fábrica y las fábricas originales son los nodos de entrada/salida de la misma.

Veamos en el ejemplo anterior los pasos del algoritmo:

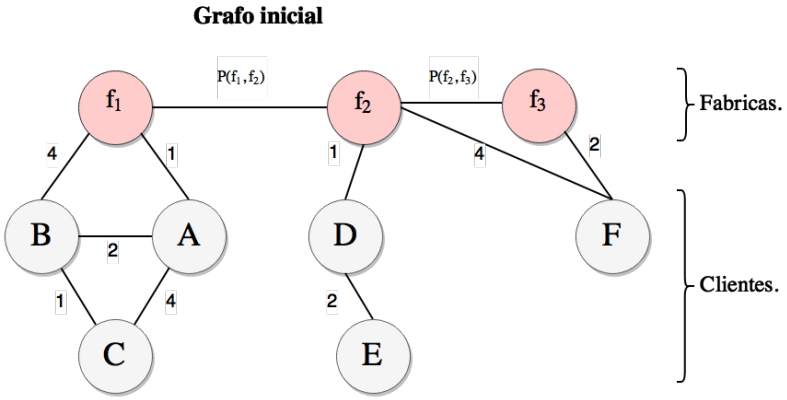


Figura 13: Grafo inicial

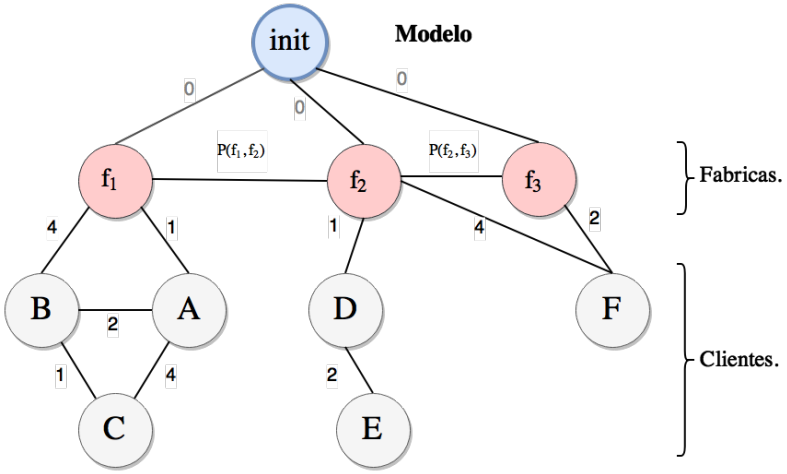


Figura 14: Se agrega el nodo ficticio.

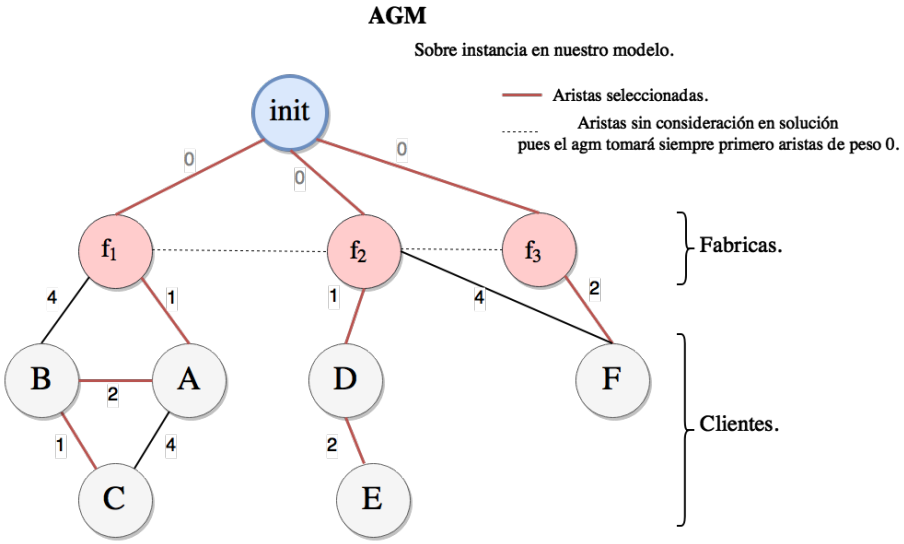


Figura 15: Árbol generador mínimo

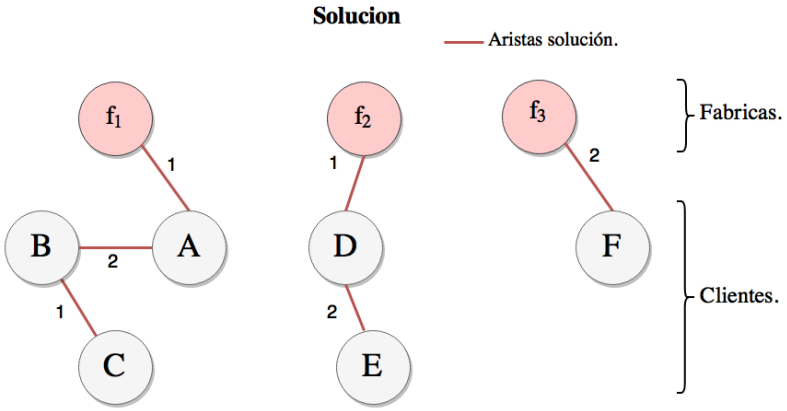


Figura 16: Solución quitándole al AGM el nodo ficticio.

3.2.1. Pseudocódigo

A continuación presentamos el pseudocódigo.

Algorithm 4: obtener_rutas(lista($aristas$))

1: $G \leftarrow$ Agregar nodo ficticio u con aristas sin peso hacia las fábricas ▷ $\mathcal{O}(F)$
2: $T \leftarrow$ **Kruskal**($aristas$) ▷ $\mathcal{O}(RlogC)$
3: **return** aristas($T \setminus \{u\}$) ▷ $\mathcal{O}(C)$

Complejidad: $\mathcal{O}(Rlog(C))$
Justificación: Para agregar el nodo ficticio hay que agregar las aristas desde él hacia todas las fábricas, eso cuesta $O(F)$. Kruskal cuesta $O(Rlog(R))$ pero como $F \leq C$ y $R \in O((F+C)^2) \in O(4C^2) \in O(C^2)$. En peor caso la complejidad resulta $O(Rlog(C^2)) \in O(Rlog(C))$. La complejidad total del ejercicio es $O(F) + O(RlogC) + O(C) = O(C) + O(RlogC) + O(C) = O(RlogC) + O(C) = O(RlogC)$.

El pseudocódigo de la función *obtener_rutas*:

- Implementa la lista de *aristas* está basada en vectores.
- Implementa *Kruskal*[5] mediante una estructura de datos *UnionDisjointSet*, tal que esta es un *conjunto* disjunto de *componentes conexas* con *representante*, en el que puede unirse componentes o saber cuál es el representante de un elemento dado. Esta estructura es utilizada para saber cuándo dos vértices pertenecen a la misma componente conexa. El *UnionDisjointSet* ha sido implementado sobre un *diccionario* de *padres* (árbol) tal que el representante de cada elemento es la raíz del árbol al que pertenece.

3.2.2. Correctitud

La correctitud de la solución propuesta depende de ciertos aspectos, de la correctitud del modelo y de la correspondencia entre el problema propuesto y la resolución aplicada mediante AGM.

Dado $G = (V, X)$ nuestro grafo original con fábricas y clientes definimos el grafo con el nodo fictioso u agregado y aristas nulas a todas las fábricas como: $G + u = (V', E')$, con $V' = V \cup \{u\}$, $E' = E \cup \{(u, f)|$ el vértice f es una fábrica$\}$ y $(\forall f)p(u, f) = 0$.

Sea T el AGM obtenido de $G + u$ con el algoritmo de Kruskal y $S = \sum_{e \in X(T)} p(e)$. Supongamos que existe una solución óptima E^*, es decir, un subconjunto de aristas del G original tal que la sumatoria de todas ellas es S^*, de modo que $S^* < S$.

Como E^* es solución del problema, existen aristas que inciden en todos los clientes, es decir todo cliente es alcanzable (en particular, existe un camino de alguna fábrica hacia el). Por lo tanto existe por lo menos una fábrica f^* tal que forma parte de la solución y podría haber fábricas que no formen parte.

Definamos entonces un nuevo grafo con las aristas de E^* pero agregándola el nodo u tal como hicimos con G, $G^* = (V', E^* \cup \{(u, f)|$ el vértice f es una fábrica$\}$ y $(\forall f)p(u, f) = 0$. De esta manera, estamos obteniendo un grafo para el cual existe camino hacia todo cliente, hacia u pues lo agregamos recién y por lo menos estará la arista (u, f^*), y hacia todas las fábricas también a través de las aristas con u. Entonces, todo vértice es alcanzable en G^*. Veamos que la sumatoria de sus aristas es:

$$\sum_{e \in X(G^*)} p(e) = \sum_{e \in E^* \cup \{(u,f), f \text{ fábrica}\}} p(e) = \sum_{e \in E^*} p(e) = S^* < S$$

Tenemos entonces que la sumatoria de las aristas de G^* es menor que la suma de las aristas de T, pero esto es **absurdo** pues T era un árbol generador mínimo y la sumatoria de sus aristas era menor frente a cualquier otro árbol generador (ó grafo generador).

3.3. Experimentación

Las experimentaciones se ejecutarán en el entorno: MacBook Pro, procesador 2.3GHz Kaby Lake i5 dual-core, 8GB RAM, sistema operativo macOS (64 bits).

3.3.1. Complejidad

En primer lugar nos interesará comparar la performance del algoritmo frente a la complejidad postulada anteriormente. Para ello, nos valemos del *coeficiente de Pearson*, dicho coeficiente nos indica qué tan fuerte

[5] http://www.jstor.org/stable/2033241

es la correlación entre dos variables cuantitativas. Cuanto más se aproxime a 1, nos indicará una mayor correlación.

Dado que la función de complejidad postulada es $\mathcal{O}(Rlog(R))$ que como vimos es igual a $\mathcal{O}(Rlog(C))$ dado que como máximo $R = C^2$, esperaremos que la correlación mas fuerte se dé con alguna de las funciones.

Para realizar dicha experimentación, dejaremos fija la cantidad de clientes y fábricas, en 51 y 49 respectivamente. Luego, iteraremos sobre la cantidad de ejes, partiendo desde un input válido, es decir un árbol (dado que tiene la mínima cantidad de ejes que lo hacen conexo), de modo que tendrá 99 ejes ($m = n$ - 1) hasta 1000 ejes. Para cada instancia se asignan pesos en los ejes de manera aleatoria. Por último, para cada valor de m (cantidad de ejes) se consideran 50 muestras, tomando luego la media de los tiempos.

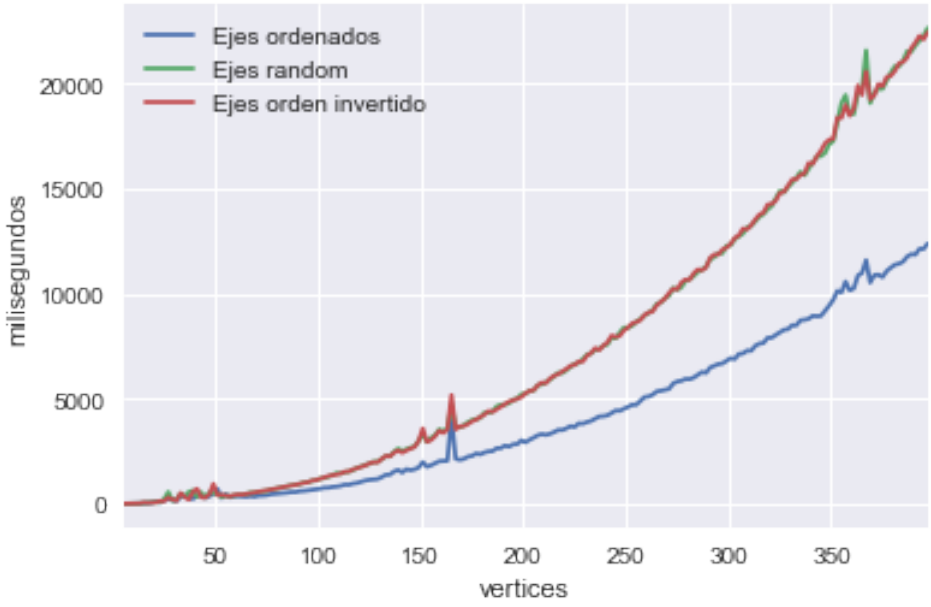
	$R * log(100)$	$R * log(R)$	$R^2 * log(100)$
nanosegundos	0.984978	**0.987000**	0.977765

Dado que la cantidad de ejes es siempre menor al máximo (n^2), y el experimento itera sobre la cantidad de ejes, era de esperarse que la correlación mas fuerte se de con $\mathcal{O}(Rlog(R))$, y así fue.

3.3.2. Segunda experimentación

Dado que la complejidad del algoritmo está atada al costo de ordenar los ejes. Nos interesa experimentar en torno los mismos. Eso significará entregar al algoritmo instancias en donde sus ejes se encuentren ordenados de distinta manera y evaluar la performance del algoritmo.

Para ello nos valemos de grafos completos, en donde iteraremos sobre la cantidad de vértices. Para cada valor de n, que toma valores entre 3 y 200, iteramos unas 100 veces, esto es naturalmente para luego tomar una media de los tiempos y descartar outliers. En cada iteración generaremos una instancia del problema, la cual serán los ejes del grafo completo, con pesos distribuidos de manera aleatoria, con dicha instancia ejecutamos nuestro algoritmo para que soluciones el problema. Luego ordenamos los ejes previamente y luego invocamos al algoritmo. Y por último los ordenamos exactamente al revés y también invocamos al algoritmo. En cada caso tomamos los tiempos que le lleva resolver el problema al algoritmo. A continuación los resultados:



Evidentemente no darles un orden particular (random), u ordenarlos de manera invertida a la esperada da lo mismo. En cambio ya brindar los ejes previamente ordenados, otorga una mejora de performance sensible. A priori no esperábamos ninguna mejora, dado que entendemos que el sort de la librería estandard de C++ no realizar ninguna consideración sobre si el arreglo estaba previamente ordenado, sin embargo debe realizar menos operaciones por si estarlo.

3.3.3. Tercer experimentación

Por último nos va a interesar entender el impacto que tiene agregar los ejes de peso 0 a nuestro grafo de entrada. Recordamos que dichos ejes se conectan desde un nuevo vértice (ficticio) a todas las fábricas.

Entonces lo que haremos sera experimentar siempre sobre un grafo completo de 1000 nodos, donde la primer iteración tendrá 900 clientes y 100 fábricas, la siguiente 899 clientes y 101 fábricas, y así hasta tener 550 clientes y 450 fábricas. De este modo hacemos crecer la cantidad de fábricas y por tanto la cantidad de ejes agregados, manteniendo siempre misma cantidad de vértices y ejes sobre el grafo de entrada. Como hicimos anteriormente para cada combinación posible de clientes y fábricas tomamos 50 muestras y calculamos la media de los tiempos obtenidos.

No se observan cambios importantes a lo largo de la ejecución del experimento, y esto se da porque la cantidad de ejes de peso 0 siempre son un número pequeño en relación a la cantidad de ejes que ya tiene el grafo de entrada. De manera que nunca se observarán diferencias importantes en la performance.