Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

Integrante	LU	Correo electrónico
Laura Muiño	399/11	mmuino@dc.uba.ar
Martín Santos	413/11	martin.n.santos@gmail.com
Luis Toffoletti	827/11	luis.toffoletti@gmail.com
Florencia Zanollo	934/11	florenciazanollo@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Ejercicio 1

1.1. Descripción:

En este ejercicio debemos encontrar el costo mínimo de un conjunto de impresiones, dado un grupo de trabajos.

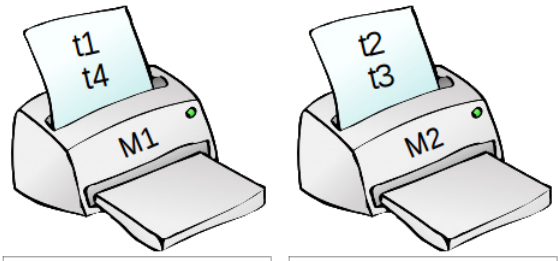
Para realizar la impresión contamos con dos máquinas iguales donde podemos repartir los trabajos, estos tienen un costo particular según quién fué el trabajo previo o si son el primer trabajo de la máquina.
Además se debe cumplir un orden en las impresiones, el trabajo i no puede ir antes que el j si i>j.
Es decir, los trabajos se pueden repartir entre las máquinas de muchas maneras distintas, pero para calcular el costo debe tenerse en cuenta el orden mencionado anteriormente.

Es nuestra meta encontrar la forma de repartirlos que abarate el costo total de las impresiones.

Por ejemplo si contamos con estos trabajos i, con sus respectivos precios respecto al trabajo anterior j:

ti \ tj	0	1	2	3
1	1	x	x	x
2	2	4	x	x
3	6	3	1	x
4	3	2	5	4

Entonces la solución con costo mínimo sería:



Costo M1:
t1 primero +
t4 despues de t1 =
1 + 2 = 3

Costo M2:
t2 primero +
t3 despues de t2 =
2 + 1 = 3

Costo total:
M1 + M2 = 6

1.2. Pseudocódigo:

Algorithm 1 Impresiones ordenadas

```
1:  procedure DIVIDIR TRABAJOS(n, Costos)   ▷ n= cant trabajos, Costos= costo de cada trabajo respecto a cada posible anterior
2:      Inicializo una matriz dp, tamaño n*n
3:      dp[0][1]← Costos(1, 0)   ▷ en la posición (0,1) de la matriz guardo el valor de hacer el trabajo uno primero
4:      agregar_a_maquina(1)   ▷ guardo en qué máquina agregué el trabajo
5:      for j ← 2, n do   ▷ el trabajo 1 ya lo agregué
6:          for i ← 0, j − 1 do
7:              if i = j − 1 then
8:                  dp[i][j] ← minimo(dp[i][k] + Costos(j, k), ∀ 0 ≤ k ≤ j − 2)
9:                  agregar_a_maquina(j)   ▷ Costos(j, k) es el costo de hacer el trabajo j después del k
10:             else
11:                 dp[i][j] ← (dp[i][j − 1] + Costos(j, j − 1))
12:                 agregar_a_maquina(j)
13:             end if
14:         end for
15:     end for
16:     return minimo(dp[i][n], ∀ 0 ≤ i ≤ n − 1)
17:                 ▷ retorno el mínimo de la última columna, además de la combinación para lograr ese mínimo
18: end procedure
```

Nota: $agregar\_a\_maquina(j)$ modifica un entero que depende del $agregar\_a\_maquina(j-1)$ pero sigue siendo O(1) ya que accede una vez a la matriz, realiza una comparación de enteros y modifica una variable entera.

1.3. Complejidad:

Para analizar la complejidad voy a intentar representar cómo 'funciona' el algoritmo con una entrada de n trabajos. La idea de esto es ver qué cosas calcula y de qué manera, para poder ver cuántas operaciones realiza en total.

Me pareció mejor esta forma en vez de un análisis linea por linea ya que se ve más detalladamente cómo actua el algoritmo. Además para no mezclar tantas cosas y que sea más legible voy a olvidarme de las partes de $agregar\_a\_maquina(j)$ (Líneas 4, 9 y 12 del pseudocódigo), esto obviamente no influye en la complejidad total ya que cada operación de ese tipo es O(1).

En principio se crea una matríz $n * n$ y se modifica la posición (0,1) con el costo del trabajo 1. Esto es O(n^2) + O(1)= O(n^2)[1] (Líneas 2 y 3).

La posición (0,1) se calcula en O(1):

i \ j	1	2	3	4	...	n
0	O(1)					
1	x					
2	x	x				
3	x	x	x			
...	x	x	x	x		
n-1	x	x	x	x	x	[2]

[1] La matríz está implementada como vector de vectores

[2] Los casilleros en blanco son los que faltan calcular y los que contienen una x nunca se calculan porque el For de la Línea 6 va desde

Luego, ingresamos en los For (Líneas 5 y 6). El algoritmo recorre las posiciones en blanco indicadas en el gráfico (la 'mitad' de la matriz si trazaramos una linea en diagonal) y para cada posición hace lo siguiente:

Si $i = j - 1$:
El algoritmo recorre las posiciones válidas de la columna anterior a la que se encuentra procesando, y busca el mínimo entre ellas (sumandole el costo de hacer j, Línea 8).
Por lo tanto se debe encontrar el mínimo de j-1 posiciones, esto es O(j-1).

Si $i \neq j - 1$:
La posición (i,j) de la matriz necesita sólo la posición (i,j-1) para ser calculada, esto es O(1) ya que se accede a la matriz y se suman enteros (Línea 11).

En resumen si $i = j - 1$ es O(j-1) sino es O(1):

i \ j	1	2	3	4	...	n
0	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
1	x	O(j-1)	O(1)	O(1)	O(1)	O(1)
2	x	x	O(j-1)	O(1)	O(1)	O(1)
3	x	x	x	O(j-1)	O(1)	O(1)
...	x	x	x	x	O(j-1)	O(1)
n-1	x	x	x	x	x	O(j-1)

El algoritmo entonces recorre $\sum_{j=2}^{n}(j-1) + n$ posiciones donde $\sum_{j=2}^{n}(j-1)$ de ellas las resuelve en O(1) y el resto en O(j-1), que como j es a lo sumo n O(j-1)=O(n).

Puedo decir entonces que la complejidad final es hacer $\sum_{j=2}^{n}(j-1)$ veces $O(1) + n$ veces $O(n)$.[3]

Ahora $\sum_{j=2}^{n}(j-1) = \frac{n^2-n}{2}$ por lo tanto es $O(n^2)$

Complejidad del ciclo: $O(n^2) * O(1) + O(n) * O(n) = O(n^2) * O(n^2) = O(2 * n^2) = O(n^2)$
La línea 16 es O(n) (recorre la última columna de la matriz).

Entonces, complejidad total: $O(n^2) + O(n^2) + O(n) = O(n^2)$

1.4. Demostración:

La idea del algoritmo es sacar el costo según quién es el último trabajo de cada máquina (en vez de realizar todas las combinaciones) e ir guardando estos datos en una matriz para reutilizarlos más adelante.

Idea: dp(i,j)= mínimo costo para que una máquina termine en i y otra en j.

El algoritmo está definido por la siguiente función, con i<j [4]:

i=0 hasta j-1.
[3] Disculpen el abuso de notación
[4] para no sacar dos veces los valores, las máquinas son iguales por lo tanto hacer (i,j)=(j,i)

$$dp(i,j) = \begin{cases} dp(i,j-1) + Costos(j,j-1) & si \quad i \neq j-1 \\ minimo(\ dp(k,i)\ + Costos(j,k)\ \forall 0 \leq k \leq j-2\ ) & si \quad i = j-1 \end{cases}$$

Como i<j e i≠j-1 ⇒ i<j-1, entonces para que una máquina termine en i y otra en j, j debe venir después de j-1 ya que j-1 no puede ir antes que i ni de ningún otro que no sea j (j-1 es el más grande distinto de j).

Si i=j-1 puedo combinar los trabajos de muchas maneras y j puede venir después de 0,1,... j-2. Además el costo de hacer j después de cada uno de ellos puede variar mucho, entonces debo ver todas esas combinaciones y luego puedo quedarme con la mínima que es la que me interesa.

Para demostrar la correctitud voy a probar que dada una posición (i,j) de la matriz, tengo en ella el mínimo costo de todas las combinaciones de forma que en una máquina el último trabajo sea i y en la otra sea j.

Hipótesis inductiva:
Dada una columna j de la matriz, en cada posición (i,j) con 0≤i<j, tengo el mínimo costo de todas las combinaciones de forma que en una máquina el último trabajo sea i y en la otra sea j. De ahora en más lo voy a llamar mínimo(i,j).

Caso base:
El caso de j=1 es trivial y se encuentra fuera de los For (dónde implemento la función explicada más arriba), es un caso muy particular así que mi caso base será j=2.

Quiero ver que $\forall\ 0 \leq i < j$, dp(i,2) es mínimo(i,2).
Lo divido en casos:

Si i=0 (i≠j-1): dp(0,2) = una máquina no tiene ningún trabajo y la otra termina en t2, es decir, tiene a t1 y t2. Como existe una única forma de hacer esta combinación el resultado es mínimo(0,2).

Si i=1 (i=j-1): dp(1,2) = una máquina tiene al t1 y otra al t2, luego existe una única forma de hacer esto, entonces es mínimo(1,2).

Paso inductivo:
Supongo que $\forall\ 0 \leq k < j$, vale la H.I., quiero ver que vale para j.
Lo divido en casos:

Si i≠j-1: dp(i,j) =[5] dp(i,j-1) + Costos(j,j-1)
Como j-1<j entonces dp(i,j-1) es mínimo(i,j-1) por H.I. y Costos(j,j-1) es un valor único ⇒ dp(i,j-1) + Costos(j,j-1) es mínimo(i,j).

Si i=j-1: dp(i,j) =[6] minimo(dp(k,i) + Costos(j, k) $\forall 0 \leq k \leq j-2$)
Como k≤j-2, k<j entonces dp(k,i) es mínimo(i,k)[7] por H.I.
Como los Costos(j, k) pueden variar mucho según el k no puedo asegurar que dp(k,i) + Costos(j, k) es mínimo(i,j). Pero dentro de todos esos valores me quedo con el mínimo entonces sé que vale.

1.5. Casos de prueba:

Dado que el algoritmo debe encontrar la división de trabajos que dá el mínimo, las instancias que permiten verificar su correctitud son las siguientes:

- Hay sólo un trabajo. Ej:

ti \ tj	0
1	3

  Solución: t1 en una máquina, la otra libre, costo: 3.
  El algoritmo dá la solución esperada.

El mínimo está dado por la siguiente distribución:

[5] por definición de la función
[6] por definición de la función
[7] cómo las máquinas son iguales mínimo(i,k) = mínimo(k,i), lo escribo así para que se entienda mejor

- Todos los trabajos en una máquina y ninguno en la otra.

Ej:

ti \ tj	0	1	2	3
1	1	-	-	-
2	3	2	-	-
3	6	3	2	-
4	3	2	1	3

Solución: t1, t2, t3, t4 en una máquina, la otra libre, costo:8.
El algoritmo dá la solución esperada.

- División (en orden) de los trabajos.

Ej:

ti \ tj	0	1	2	3
1	1	-	-	-
2	1	3	-	-
3	2	3	1	-
4	2	6	5	3

Solución: t1 en una máquina, t2, t3, t4 en la otra, costo: 6.
El algoritmo dá la solución esperada.

- División (sin orden) de los trabajos.

Ej:

ti \ tj	0	1	2	3
1	1	-	-	-
2	1	4	-	-
3	2	2	3	-
4	2	3	2	4

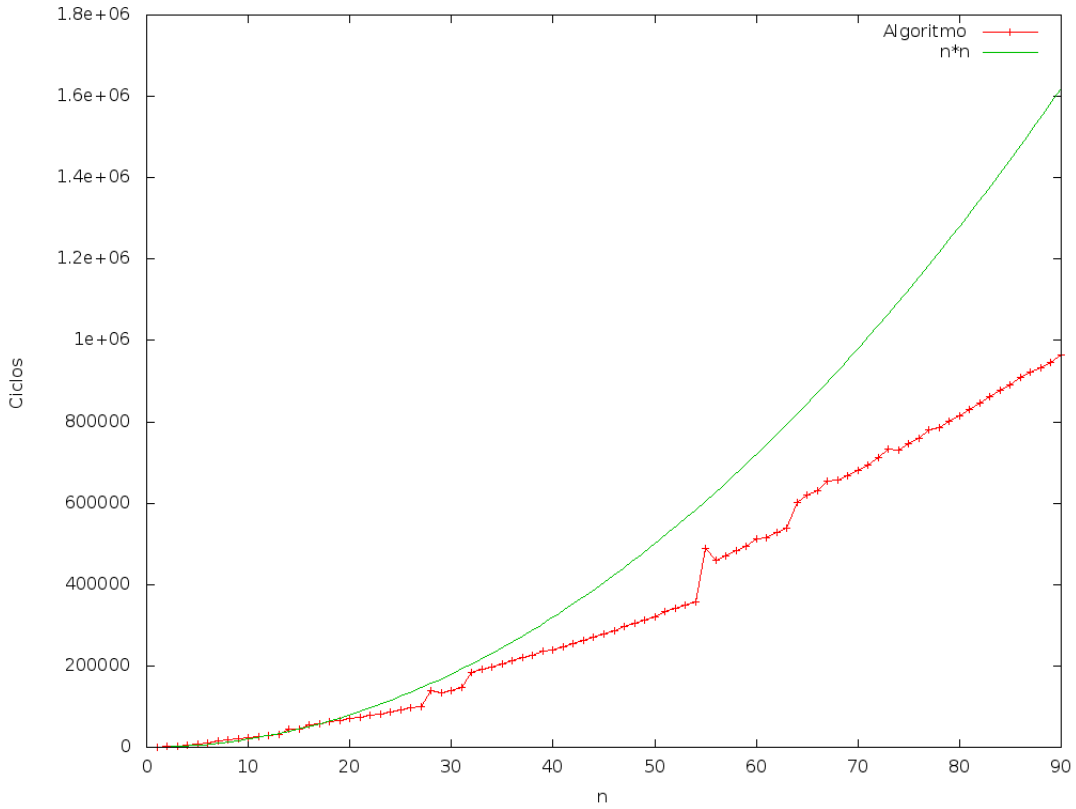
Solución: t1, t3 en una máquina t2, t4 en la otra, costo: 6.
El algoritmo dá la solución esperada.

Elegimos estos casos porque son distintas formas de distribuir los trabajos, lo que intentamos mostrar es que el algoritmo no deja distribuciones sín mirar.

1.6. Experimentación:

Se realizaron experimentos con instancias aleatorias, generadas utilizando la función rand()[8] perteneciente a la Standard Library.

Los experimentos se ejecutaron varias veces, para instancias diferentes, contando la cantidad de ciclos que conlleva resolver el problema y luego se promediaron los datos obtenidos.



Dada la naturaleza del algoritmo, no existe un peor caso. Ya que siempre debe llenar una matriz n^2 y buscar el mínimo dentro de la última columna.

Conclusiones:

- La cota calculada teóricamente era correcta.
- La técnica de programación dinámica es muy efectiva en ciertos casos.

[8] http://www.cplusplus.com/reference/cstdlib/rand/

2. Ejercicio 2.1

3. Ejercicio 2.2

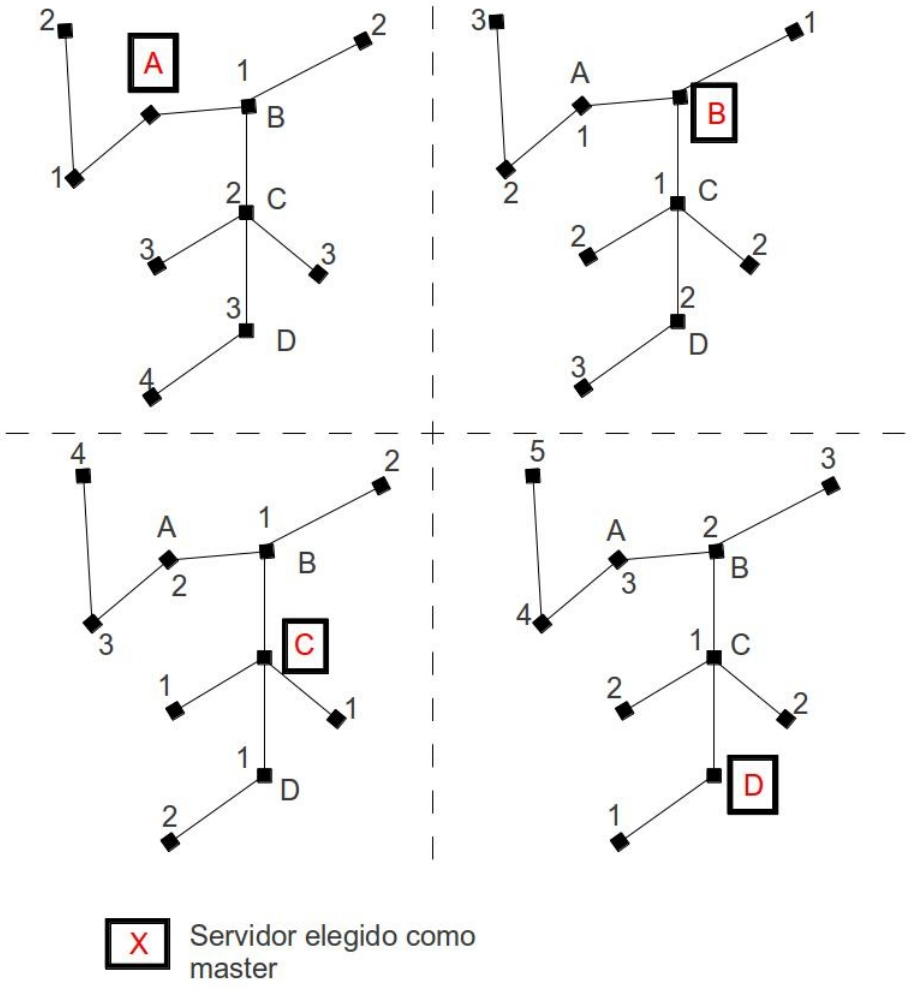
3.1. Descripción:

Recibimos una red de servidores interconectados por enlaces, nos piden encontrar cual servidor debemos escoger como master (servidor que recibe los datos primero), de tal forma que se pueda actualizar la información en todos los sistemas en el menor tiempo posible. Se sabe que se tarda el mismo tiempo de ir de servidor en servidor para todos los enlaces, y las copias dentro del mismo son instantáneas. Además cada servidor transmite simultáneamente a todos los vecinos elegidos.

En resumen el proceso es:

- Un servidor recibe la información por primera vez (master).
- Puede copiar instantáneamente la información para enviarla a todos los servidores con los que se conecta.
- Envía la información, y éstos la reciben al mismo tiempo, e inician el proceso de copiado para enviarlo a través de sus enlaces, y asi sucesivamente hasta que todos tengan su copia.

En el siguiente ejemplo vemos una red de servidores (nodos), y sobre cada uno la cantidad mínima de pasos que precisaría para copiar la información a ese servidor, tomando como master el nodo resaltado (A, B, C, D). Suponiendo que cada servidor copia a todos sus adyacentes en el momento en que recibe los datos.



Vemos que si elegimos A o C, es posible completar la transferencia en 4 pasos, si elijo el D, en 5, y si escojo el B en sólo 3, por lo tanto este es nuestro "master"óptimo (la transmisión se hace a todos los adyacentes cada vez). En este caso la solución óptima fue una sola, pero pueden existir varias, un ejemplo sencillo es una red con sólo dos servidores conectados entre sí, podría poner a cualquiera como master.

3.2. Pseudocódigo

Del enunciado del problema se pueden deducir ciertos datos que nos van a ayudar a dar con el algoritmo que buscamos.

Aunque resulte un poco obvio a esta altura, la forma de interpretar el problema es por medio de un grafo, en principio no orientado, donde los nodos representan servidores y las aristas los enlaces entre sí.

Sabemos por el enunciado que la entrada que recibimos, es la salida del ejercicio anterior, por lo tanto podemos afirmar:

- Desde un servidor (cualquiera) se debe poder llegar a todos los demás.
- La cantidad de enlaces usados es mínima.

De lo primero se deduce que el grafo es conexo, no importa que par de nodos elija, existe un camino que los comunica.

El segundo punto nos permite afirmar que no hay circuitos simples (camino que salga de un nodo y vuelva al mismo, pasando a lo sumo una vez por cada arista). Cada nodo dentro del circuito tendría al menos dos formas de conectarse con otro nodo cualquiera entonces podría sacar una arista adyacente que pertenezca a uno de esos caminos y el grafo seguiría estando conexo.

Entonces tenemos un grafo conexo sin ciclos simples, es decir un árbol. Ahora estamos más cerca de la resolución.

Ahora como sabemos que es un árbol tenemos un único camino que conecta a cada nodo con el posible master. Y lo que buscamos es elegirlo de tal forma que se pueda llegar a todos los nodos en la menor cantidad de pasos posibles. El nodo master tiene que cumplir que su distancia máxima[9] sea la mínima de entre todos los nodos.

Encontrar este nodo mediante la definición de arriba puede resultar muy costoso, ya que deberíamos calcular la máxima distancia a todos los nodos desde cada nodo, que nos conduce a una complejidad cuadrática.

En cambio vamos a resolverlo intentando encontrar un nodo que pertenezca a un máximo camino posible dentro del árbol, y además sea el centro de dicho camino, es decir la distancia desde él a ambos extremos no puede diferir en más de 1. Más adelante vamos a demostrar porque es correcto.

Para implementar el algoritmo utilizamos lo siguiente:

1. Una estructura para los nodos, que se compone de:
   - Nodos_Adyacentes: Una lista de adyacentes al nodo.
   - Prim_Long_Max y Seg_Long_Max: las longitudes de sus dos ramas más largas.
   - Nodo_Rama_Max: el nodo hacia el que está volcada la rama más larga.
   - Suma_Longs: la suma de ambas longitudes.
2. Una lista de nodos(estructura del item anterior), llamada árbol.
3. Una función recursiva que parta de un nodo y vaya calculando los datos mencionados en el primer item, haciendo llamados a los adyacentes (salvo a aquel del que vino).
4. Finalmente con los datos almacenados en la lista de nodos, busco el nodo master.

Suponiendo que ya tenemos el arreglo de n nodos, cada uno en la posición que le corresponde según el número, con la información de sus adyacentes ya cargada, pasamos a ver el pseudocódigo del algoritmo recursivo y luego del que busca el master.

Nota: Dentro del algoritmo recursivo la función ActualizarMaximos actualiza los valores de las dos ramas más largas, y hacia donde está la rama más larga. Después del ciclo se usa una función llamada ActualizarArbol que actualiza los datos de la lista de nodos, para el nodo en el que se encuentra (dos ramas largas, suma de ramas y hacia donde va la rama más larga).

[9] es la distancia máxima desde el nodo hacia todos los demás mediante caminos simples

Algorithm 2 Algoritmo Recursivo

```
1: function CalcularRamaLarga(nroNodo, nroPadre)  ▷ El primero es el ID del nodo que quiero calcular y el segundo me dice de donde proviene
2:     while QuedanHijos(Arbol[nroNodo]) do  ▷ cicla mientras todavía queden adyacentes por recorrer
3:         hijo ← ObtenerUnHijo(Arbol[nroNodo])  ▷ toma un nodo adyacente y lo guarda en Hijo
4:         if hijo! = NroPadre then
5:             rama = CalcularRamaLarga(hijo, nroNodo)  ▷ llamado recursivo
6:             ActualizarMaximos(rama, ramaMax1, ramaMax2, nodo_rama_Max1)  ▷ ver *Nota*
7:         end if
8:     end while
9:     ActualizarArbol(nroNodo, ramaMax1, ramaMax2, nodo_rama_Max1)  ▷ ver *Nota*
10:    return(ramaMax1 + 1)  ▷ esto devuelve al padre la longitud de la rama más larga hacia este nodo
11: end function
```

Algorithm 3 EncontrarMaster

```
1: procedure BuscarMaster
2:     nodo ← Max_Suma(Arbol)  ▷ recorre arbol, buscando el nodo con suma de ramas máxima
3:     r1 = Arbol[nodo].ramaMax1  ▷ en r1 y r2 tengo las longitudes de las ramas más largas del nodo
4:     r2 = Arbol[nodo].ramaMax2
5:     while r1 != r2 AND (r1 -1) != r2 do  ▷ cicla mientras la diferencia entre r1 y r2 sea mayor a 1
6:         nodo = arbol[nodo].nodoRamaLarga  ▷ me muevo hacia el nodo con la rama más larga
7:         r1 − −  ▷ siempre me muevo hacia la rama de r1
8:         r2 + +
9:     end while  ▷ Cuando salgo del ciclo nodo contiene al master
10: end procedure
```

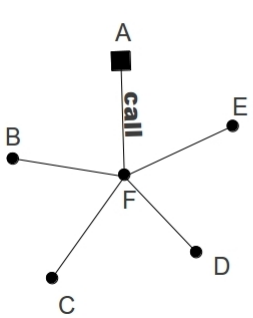
3.3. Complejidad

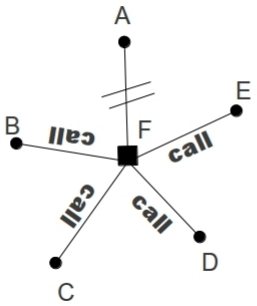
Para encontrar la complejidad de nuestra solución, tenemos que analizar dos algoritmos, uno recursivo y el otro que busca el master, ambos con sus respectivos pseudocódigos detallados en el punto anterior.

En ambos algoritmos, tenemos un arreglo de nodos 'arbol', acceder a uno de ellos con un índice me cuesta O(1).

En el primero tenemos un ciclo, y dentro de él un llamado recursivo. El algoritmo no vuelve a pasar por un nodo en el que estuvo, es decir arribo a cada nodo por exactamente una de las aristas y luego me muevo hacia las demás. De esta manera encuentro una cota para la cantidad de llamados recursivos, que es n el número de nodos.

En un llamado a la función, el ciclo realiza una cantidad de iteraciones igual a la cantidad de nodos adyacentes y realiza un llamado en todos excepto aquel por el que vino. Esto significa que en toda la ejecución, cada arista es atravesada por dos nodos (pero el llamado recursivo se hace en uno de los dos), y como el número de aristas es n-1, puedo decir que el número de iteraciones del ciclo en toda la ejecución está acotado por 2n.





En el gráfico de la izquierda, la función está en el nodo A, su único nodo adyacente es F, así que realiza un llamado a él.
En el esquema de la derecha llegamos al nodo F, tiene 5 adyacentes, pero realiza el llamado recursivo a B, C, D y E, pero descarta el A, en la búsqueda de adyacentes lo tiene en cuenta pero no realiza el llamado porque proviene de ahí.

Se puede ver el "doble recorrido" sobre cada arista y que no puede existir una tercera "pasada".

De esta manera puedo afirmar que el ciclo itera a lo sumo 2n veces, y lo que está fuera de él n veces (En total durante toda la ejecución)

Análisis del algoritmo recursivo:

Línea 2: Preguntar si quedan "hijos."es decir nodos adyacentes, es **constante** ya que tengo un arreglo Arbol y el índice en el arreglo que cuesta O(1) acceder y luego empiezo a recorrer la lista con los adyacentes.

Línea 3: Tomo cualquiera de los hijos de la lista, es **constante**.

Líneas 4 y 5: Acá se hace el llamado recursivo, tiene complejidad **constante** hacer el llamado, ya tengo acotada la cantidad de llamados así que solamente hago el análisis de las operaciones que empleo en la función. El if también es de complejidad **constante**.

Línea 6: La función que uso en esta línea realiza un par de comparaciones para determinar los dos valores mayores de los tres que recibe. Complejidad **constante**.

Línea 8: Fin del ciclo, ya tengo acotada la cantidad de iteraciones totales (suma de iteraciones entre todos los llamados) así que el análisis de iteraciones lo dejo para después por ahora sólo digo que tengo una suma de complejidades constantes, así que la complejidad del ciclo (sin tener en cuenta iteraciones) es **constante**.

Líneas 9 y 10: Actualiza la información del nodo en el arreglo con los valores definitivos, y devuelve un valor. También complejidad **constante**.

Por un lado tengo que la cantidad de llamados a la función durante toda la ejecución está acotada por n, entonces las líneas 9 y 10 se ejecutan a lo sumo n veces, es decir que tenemos O(1) x n = O(n). Además sé que el ciclo que tiene complejidad constante para una iteración, durante toda la ejecución itera a lo sumo 2n veces, obtenemos O(1) x 2n = O(2n).

De esta manera para calcular la complejidad del algoritmo en una ejecución, debo sumar las complejidades de ambos. O(n) + O(2n) = O(3n) = O(n)

Nos queda ver la complejidad del segundo algoritmo, se ejecuta una única vez, por lo tanto con la suma de las complejidades de cada una de sus operaciones, voy a obtener la complejidad total del algoritmo, que luego sumada a la complejidad que calculé para el algoritmo 1, obtengo cual es la complejidad de mi resolución del problema.

Análisis del Algoritmo 2, para encontrar el master:

Línea 1: Recorre el arreglo 'arbol' buscando el elemento que posee el mayor campo suma. A lo sumo recorro todo el arreglo por lo tanto tengo una complejidad en el orden **O(n)**.

Líneas 2 y 3: Obtengo valores del arreglo, accediendo con un índice. Complejidad **constante**.

Línea 4: Tenemos un ciclo que itera una cantidad de veces igual a la mitad de la longitud del máximo camino en un árbol. Un camino máximo puede ser de a lo sumo n-1 aristas, así que puedo acotar el número de iteraciones en n veces.

Línea 5: Acceso a un campo de un elemento del arreglo, con un determinado índice. Complejidad **constante**.

Líneas 6 y 7: Sumo y resto valores, complejidad **constante**.

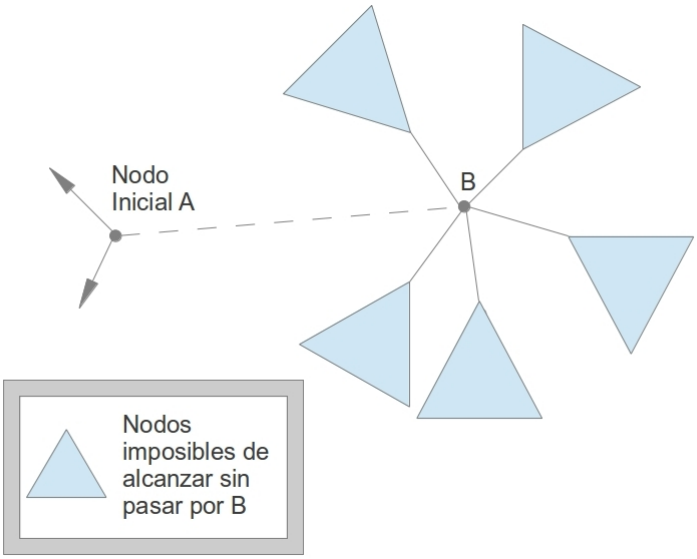
Entonces tenemos un ciclo que itera n veces, dentro del cual hay operaciones de complejidad constante, ésto es O(1) x n = O(n). Y en la línea 1 teníamos un recorrido por el arreglo en el orden de O(n), por lo tanto este algoritmo tiene una complejidad de O(n) + O(n) = O(2n).

Complejidad total (algoritmo 1 + algoritmo 2): Ahora ya tengo las complejidades de ambos, las sumo y obtengo el total: O(n) + O(n) = O(2n) = O(n)

Queda demostrado que nuestra solución se encuentra dentro de la complejidad exigida por el enunciado.

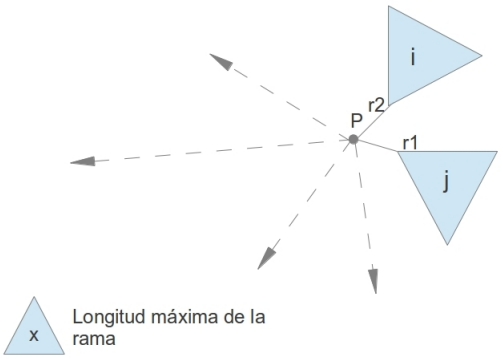
3.4. Demostración

Aclaración: Al ser un árbol, sabemos que hay una única forma de llegar de un nodo A a uno B (pasando una vez por cada arista) por lo tanto si B tiene k ramas adyacentes, en el momento que llego a B (recorriendo siempre hacia un adyacente) sólo una de esas ramas puede estar explorada (en las demás aún no pude alcanzar ningún nodo). Si hubiese ya alcanzado algún nodo en una de las otras ramas, entonces si sigo recorriendo esa rama lo podría alcanzar nuevamente e ir desde allí hacia el nodo donde comencé generando un ciclo simple, lo cual es absurdo porque en un árbol no tengo ciclos simples.



Dado un árbol de n nodos, elijo un nodo P cuyas dos ramas más largas conforman un camino de longitud máxima del árbol (r1 tiene una longitud de h y r2 una longitud de i con i $\leq$ h, i puede ser cero), además estas ramas difieren en a lo sumo una arista de longitud (entonces i = h-1 o i = h).

De P se desprenden k aristas (k <n) y aquellas que no son r1 y r2 sabemos que tienen una longitud máxima menor o igual a r2. Entonces elegir a P como master me resuelve el problema en h pasos (h $\geq$ i $\geq$ resto de ramas) además h + i = longitud de camino máximo.



Ahora supongo que existe otro nodo P' que me resuelve el problema en menos pasos, entonces llegar al nodo P me va a llevar cierta cantidad de pasos j, con j $\geq$ 1, y en el momento de llegar a P al menos una de las dos ramas (r1 o r2) no puede haber sido atravesada en absoluto, es decir en alguna de las ramas ningún nodo fue alcanzado todavía (ver Aclaración del comienzo).

Por lo tanto al menos voy a necesitar recorrer alguna de ellas para completar el recorrido por todos los nodos del árbol, suponiendo que estoy en mi mejor caso y que me queda recorrer la rama r2 que tiene longitud menor o igual a r1, le sumo los pasos que me costó llegar a P nos queda:

como j $\geq$ 1 y long(r2)=i $\Rightarrow$ i + j, además sé que i= h-1 o i=h $\Rightarrow$ h - 1 + j $\geq$ h ó h + j >h

Así la cantidad de pasos no puede ser menor si elijo P', a lo sumo es igual. Por lo tanto P está bien escogido, aunque no sea la única solución.

La implementación encuentra un nodo de un camino máximo y luego haciendo el 'balanceo' llego a un master óptimo.

Lo que tengo que demostrar es que empezando desde un nodo C cualquiera y recorriendo sus adyacentes (sin volver nunca por donde vine) en algún momento entro en un camino máximo y la longitud del mismo la puedo calcular como la suma de las dos ramas más largas del nodo (sin contar la rama por la que vine, y teniendo en cuenta la aclaración).

Tengo dos casos, que el nodo donde arranco no pertenezca a un camino máximo, o que pertenezca.

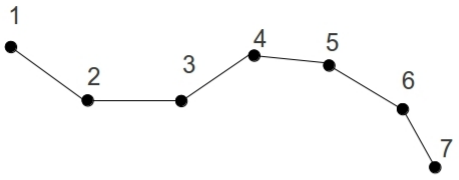
- Si no pertenece, en algún momento llego a un nodo M que si pertenece, si sumo las dos ramas más largas de M voy a tener la longitud del camino máximo. Puedo afirmar ésto porque la rama por la cual llego a M, no puede tener una longitud mayor a ninguna de sus dos ramas más largas, ya que si fuera así el camino máximo tendría que involucrar al nodo anterior por el que ya pasé adyacente a M, pero es falso porque dijimos que M era el primer nodo que alcanzamos que pertenecía a un camino máximo.
- Si arranco en un nodo perteneciente a un camino máximo, la suma de sus ramas más largas es igual a la longitud de dicho camino. Esto es verdad siempre, si r1 y r2 son mis ramas más largas, longmax(r1)≥ longmax(r2)≥ otras ramas, el camino máximo que pasa por este nodo es igual a longmax(r1) + longmax(r2).

Ya vimos que requisitos cumple un master óptimo, y luego que la implementación encuentra un nodo de un camino máximo, falta un último detalle y es el 'balanceo'. El nodo que encontramos pertenece al camino máximo, y moviendome hacia el nodo adyacente con la rama más larga (siempre me mantengo en el camino máximo) en algún momento tengo que encontrar un nodo cuyas dos ramas más largas sean iguales o difieran en una arista de longitud. La forma en que se implementa se puede ver en el pseudocódigo del algoritmo EncontrarMaster, o en el código fuente.

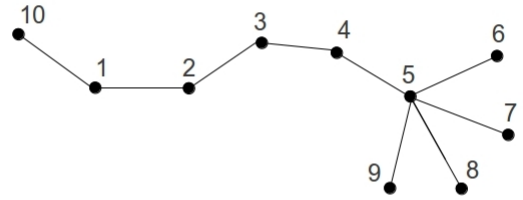
3.5. Casos de Prueba

Vamos a presentar 5 instancias distintas del problema mediante gráficos, y la solución que nos devuelve la implementación, de manera de hacer una comprobación visual del funcionamiento. Más abajo vemos las soluciones posibles y lo que nos devolvió el algoritmo. Los ejemplos abarcan casos que tienen 2 soluciones y otros sólo una, y árboles con distintos grados máximos.

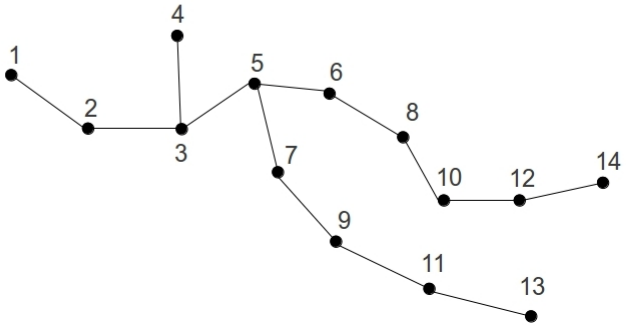
1)



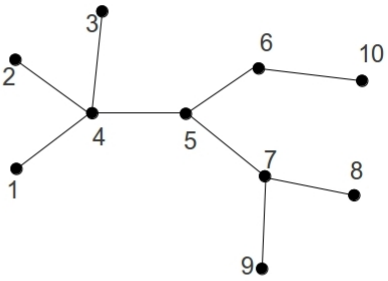
2)



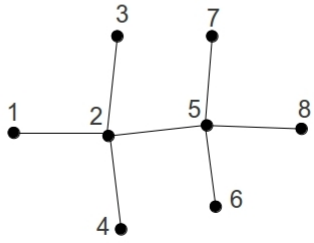
3)



4)



5)



Grafo	Solución posible (nodo)	Solución dada por el algoritmo
1	4	4
2	3	3
3	6 ó 5	5
4	5	5
5	2 ó 5	2

4. Ejercicio 3

4.1. Descripción

Disponemos de cierta configuración de rutas que comunican distintas ciudades. En estas ciudades podemos encontrar tanto clientes como fábricas. Cada ruta tiene un costo de inversión proporcional a la longitud de la misma. Debido a que las rutas no se encuentran preparadas para afrontar el paso de camiones, se nos pide que fortalezcamos algunas de ellas de manera tal que cualquier cliente pueda ser provisto por alguna de las fábricas. La solución debe ser mínima en cuanto al gasto de inversión asociado. Se sabe que es posible satisfacer la demanda de cada cliente y que no hay más fábricas que clientes. A continuación un ejemplo del problema y una solución posible:

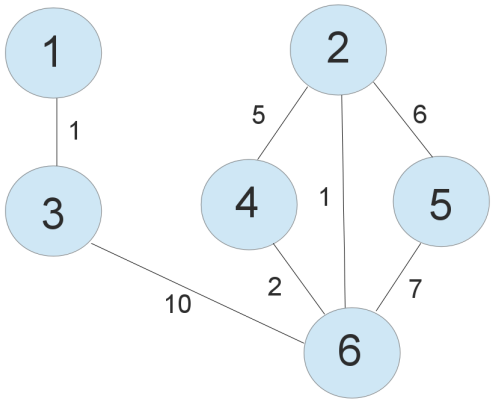


Figura 1: En este ejemplo disponemos de 2 fábricas y 4 clientes. Los nodos < 3 representan fábricas y los ≥ 3 representan clientes. El largo de cada ruta se detalla al lado de cada arista.

Solución:

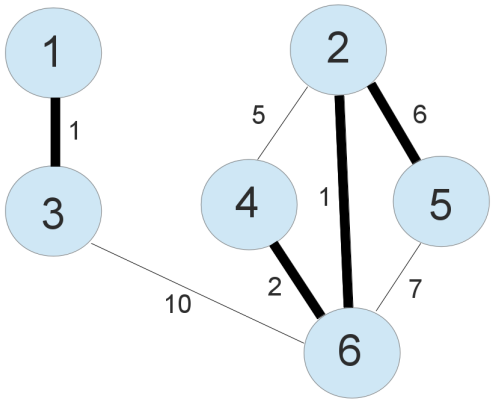


Figura 2: Solución posible. Las rutas marcadas en negro se fortificarán.

Cada cliente de la solución hallada es provisto por alguna de las fábricas disponibles y el costo total de inversión es minimal: 10.

4.2. Implementación y Complejidad

Para implementar Prim modificado utilizamos una cola de prioridad de manera tal que la elección de las aristas resultantes sea eficiente. A continuación un pseudocódigo de la implementación basado en el pseudocódigo presentado

anteriormente pero de más bajo nivel.

Algorithm 4 PrimModificado

```
1: procedure PrimModificado(G = (V, E), int cantClientes, int cantFabricas)
2:     T ← ∅
3:     Q ← priority_queue < arista >
4:     for all e = (u, v) ∈ E/ u es una fábrica y v no do        ▷ recorro todas las fábricas y busco sus adyacentes
5:         push(e, Q)
6:     end for
7:     for i ← 0 to cantClientes do
8:         e = (u, v) = obtenerMin(Q)
9:         T ∪ {e}
10:        for all j ∈ V / v es adyacente a j con j ≠ u do
11:            push((v, j), Q)
12:        end for
13:    end for
14:    return T
15: end procedure
```

Algorithm 5 obtenerMin

```
1: procedure obtenerMin(G = (V, E), priority_queue Q)
2:     bool encontre = false
3:     res = (u, v)
4:     e' = (u', v') = Q.pop()
5:     while ¬encontre do
6:         if v' es una fábrica y u' no tenía asociada una fábrica then
7:             encontre = true
8:             res = e'
9:         else
10:            if v' es cliente y tanto u' como v' no tenían asociadas fábricas then
11:                encontre = true
12:                res = e'
13:            end if
14:        end if
15:    end while
16:    return res
17: end procedure
```

La complejidad de Prim Modificado dependerá de la estructura utilizada para ir eligiendo aristas. El cuerpo del for se ejecuta C veces (líneas 7-12). El obtener mínimo dentro del while se ejecutará O(R) veces en total ya que en los C llamados como mucho obtiene R aristas. Cada operación de push, pop cuesta dentro de obtenerMin cuesta log(R) lo que equivale a O(log $(F+C)^2$) ya que $(F+C)^2$ es la mayor cantidad de aristas posibles. Como F es menor que C (enunciado), log($(C+F)^2$) $\subset$ log(C^2). Por lo tanto la complejidad equivale a O(2 log C) $\subset$ O(log C). El for que se encarga de pushear las aristas adyacentes a la cola de prioridad (líneas 10-12) también se ejecutará un total de R veces y la operación de push cuesta log(R) $\subset$ O(log C). Por lo tanto la complejidad total será de O(R*log(C)+R*log(C)) $\subset$ O(R*log(C)) como pedía el enunciado.

4.3. Demostración

Diremos que un conjunto de aristas factible es *prometedor* si se puede extender para producir no sólo una solución, sino una solución óptima para nuestro problema. Si un conjunto de aristas prometedor ya es solución entonces esta solución debe ser óptima.

La demostración de la correctitud de algoritmo es por inducción sobre el número de aristas que hay en el conjunto T. Demostraremos que si T es prometedor en alguna fase del algoritmo, entonces sigue siendo prometedor al añandir

una arista adicional. Cuando se detiene el agoritmo (T tiene C cantidad de aristas), T da una solución prometedora al problema y por lo tanto óptima.

Caso base: el conjunto vacío T es prometedor Paso inductivo: supongamos que T es prometedor justo antes de que el algoritmo añada una nueva arista e al conjunto T. T es un conjunto de aristas prometedor por hipótesis inductiva y e es por definición una de las aristas más cortas que salen de B. Podemos asegurar que existe un e que cumpla dichos requisitos ya que el enunciado asegura que todos los clientes pueden ser provistos por al menos una fábrica. Por lo tanto T $\cup$ e también es prometedor.

Como el conjunto T es prometedor en todas las fases del algoritmo también lo será cuando finalice ya que ofrece una solción óptima del problema.

4.4. Experimentación

Para medir empiricamente la eficiencia del algoritmo, decidimos tomar varias instancias del problema con grafos G=(V,E) cuyo #V va desde 5 a 100. Graficamos los resultados de los ciclos insumidos en función del número de aristas presentes en el grafo.

Para grafos completos: