Tarea 2

Diego Méndez Medina

1. Considera un sistema distribuido representado como una gráfica de tipo anillo, cuyos canales son bidireccionales, con $n = mk$ procesos, con $m > 1$ y k impar. Los procesos en las posiciones $0, k, 2k, ..., (m-1)k$ son macados inicialmente como líderes, mientras que procesos en otras posiciones son seguidores. Todos los procesos tienen un sentido de dirección y pueden distinguir su vecino izquierdo de su vecino derecho, pero ellos no tienen información alguna acerca de sus *ids*.

   El algoritmo 1 está destinado a permitir que los lideres recluten seguidores. No es difícil ver que todo seguidor eventualmente se agrega a sí mismo a un árbol enraízado con padre en algún líder. Nos gustaría que todos esos árboles tuvieran aproximadamente el mismo número de nodos.

   - ¿Cuál es el tamaño mínimo y máximo posible de un árbol?
   - Dibuja el resultado de una ejecución para e algoritmo con $k = 5$ y $m = 4$

Algoritmo 1 Algoritmo de reclutamiento para el problema 1

Inicialmente hacer
1. **if** yo soy un íder **then**
2. parent $\leftarrow$ id
3. **send**($<$ *recluta* $>$) a ambos vecinos
4. **else**
5. parent $\leftarrow \perp$
6. **end if**

Al recibir recluta desde p **hacer:**
7. **if** parent $= \perp$ **then**
8. parent $\leftarrow p$
9. **send**($<$ *recluta* $>$) a mi vecino que no es p
10. **end if**

Solución:

Que el número de procesos sea par o no depende de m, pues una condición es que k sea impar. El primer caso del algoritmo es cuando $k = 1$ y $m = 2$, de forma que $\{0, 1\}$ son los procesos, donde por definición $\{0, (m-1)\cdot k = 1\}$ son los lideres, de forma que para el tiempo uno ya terminamos, con el arbol con raíz 0 y otro con la raíz 1. ¿Es esté el arbol, de todos los posibles, el que tiene menor altura?

Sí, por que un árbol menor sería sin procesos, lo cúal las especificaciónes no permite. En general van a existir m arboles.

Nota: el árbol antes mencionado no es el único de altura cero, considere $k = 1$ y $m = 3$. Según lo dicho debe haber tres raíces:

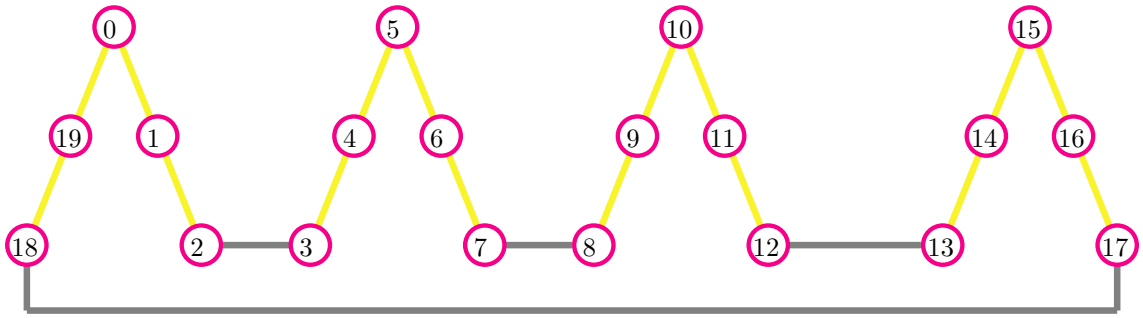
Tenemos los procesos $\{0, 1, 2\}$, de forma que dadas las instrucciónes $\{0, k = 1, (m-1)\cdot k = 2\}$ son líderes, en el tiempo uno volvemos a terminar, pero al todos tener id nadie se anexa. Así, el árbol con menor tamaño posible es el que tiene un único elemento y altura cero.

Así pues, tenemos m arboles y $m \cdot k$ procesos, de forma que para cada árbol hay k vertices. Como k siempre es impar y cada capa mayor a cero tiene dos vertices, existen $\frac{k-1}{2}$ capas por árbol. De forma que el arbol con mayor vertices y altura esta determinado por k, ya que como mencionamos m solo nos indica el número de arboles que tendran k elementos cada uno, entonces este sera cuando $m \in \mathbb{N}$ y k tienda a infinito.

Ejemplo: Cuando $m = 2$ y $k = 1001$, cada árbol tiene mil un elementos y altura quinientos.

Ejemplo: Cuando $m = 3$ y $k = 1001$, cada árbol tiene mil un elementos y altura quinientos.

Resultado de la ejecución con $k = 5$ y $m = 4$:



Donde las aristas amarillas indican que procesos son seguidores de quien, y en las grises no hubo comunicación pero siguen existiendo.

2. Realice un análisis preciso de la complejidad de tiempo y la complejidad de mensajes de :

   - El algoritmo de broadcastTree.

   Solución:

   Comenzamos enunciando el algoritmo:

   Suponemos la existencia de un árbol generador, T.

Algoritmo 2 broadcastTree(ID, soyRaiz, M):

```
1. PADRE, HIJOS, soyRaiz = soyRaiz

   Inicialmente hacer
2. Si soyRaiz entonces
3.     send(< M >) a todos los HIJOS

   Al recibir <M> de PADRE:
4.     send(< M >) a todos los HIJOS
```

En clase el profesor menciono que:

- Complejidad de los mensajes = $|V| - 1$.
- Complejidad de tiempo = $Profundidad(T)$

Lo demostraremos a continuación, sea $G = \{V, E\}$ la gráfica que tiene al árbol generador T.

Primero pensemos en la gráfica con un único vertice, llamemoslo v si queremos que el algoritmo termine entonces debe ser raíz. Así:

$$
\begin{aligned}
&u\{\\
&\texttt{Padre} : NULL\\
&\texttt{Hijos} : \emptyset\\
&\texttt{soyRaiz} : TRUE\\
&\}
\end{aligned}
$$

Al v ser el único proceso/vertice no le envía el mensaje a nadie, se cumple:

$$
\begin{aligned}
Mensajes(broadcastTree) &= |V| - 1 = 1 - 1 = 0\\
Tiempo(broadcastTree) &= Profundidad(T) = 0
\end{aligned}
$$

Generalizando:

Mensajes:

Los únicos vertices que no mandan mensajes son los que su conjunto HIJOS es vacío, es decir las hojas. Y el único vertice que no recibe mensaje es la raíz. Entonces los mensajes enviados son $V - 1$.

Tiempo:

En el tiempo cero la raíz manda mensaje a todos los elementos del árbol en la capa/profundidad uno. (Ya mostramos el caso si es que la profundidad es cero). Decimos que para el tiempo t, con $t \geq 2$, los procesos en la capa/profundidad $t - 1$ reciben y envian el mensaje.

Sea $A = \{u | profundidad(u) = max(profundidad(v)) \forall v \in V\}$, entonces A es el conjunto de procesos de hojas con mayor profundidad. Sea $d = profundidad(T)$.

Así, todos los demás procesos y en particular, en caso de existir, también las demás hojas que no comparten profundidad con cualquier $u_i \in A$, ya recibieron el mensaje. Si los procesos en la capa $d-1$ son hoja esos procesos ya terminaron, pero existe un vertice v_i para cada proceso en A tal que $d(v_i, u_i) = 1$ y, está de más decirlo pero lo repetimos, $profundidad(v_i) = d - 1$. Así en el tiempo $d - 1$ recibieron y enviaron el mensaje, con lo que en d terminamos.

Concluimos:

$$
\begin{aligned}
Mensajes(broadcastTree) &= |V| - 1\\
Tiempo(broadcastTree) &= Profundidad(T)
\end{aligned}
$$

- El algoritmo de convergecast.

Solución:

Comenzamos enunciando el algoritmo:

Suponemos la existencia de un árbol generador, T.

Algoritmo 3 convergecast():

```
1. PADRE, HIJOS, recibidos = 0

   Inicialmente hacer
2. Si |HIJOS| == 0 entonces
3.     send(< ok >) a PADRE

   Al recibir <ok> de algún puerto en HIJOS:
4. recibidos++
5. Si recibidos == |HIJOS| entonces:
6.       send(< ok >) a PADRE
```

En clase el profesor menciono que:

- Complejidad de los mensajes = $|V| - 1$.
- Complejidad de tiempo = $Profundidad(T)$

Lo demostraremos a continuación, sea $G = \{V, E\}$ la gráfica que tiene al árbol generador T.

Los algoritmos antes mencionados siempre hacian uso de la variable $soyRaiz$, si bien en este esta inicializada, no hacemos uso de ella. Lo que hace esté algoritmo es empezar con las hojas, que por definición su conjunto HIJOS es vacío, y de ahí subir. De nuevo comencemos viendo si para la gráfica con un unico proceso se cumple. Sea u el único proceso, entonces en el tiempo cero el estado de u es:

$$\begin{aligned} u\{&\\ &\texttt{Padre} : NULL\\ &\texttt{Hijos} : \emptyset\\ &\texttt{soyRaiz} : TRUE\\ &\texttt{recibidos} : 0\\ &\} \end{aligned}$$

Su padre es NULL, si bien el algoritmo no especifica que en caso de ser raíz y $|HIJOS| == 0$ o $recibidos == |HIJOS|$ deberiamos terminar, entendemos que al ser padre vacio no se envía mensaje o más bien nadie lo recibe. En el tiempo 0 se cumple que $|HIJOS| == 0$, con lo que no hay una siguiente ejecución. Se cumple:

$$\begin{aligned} Mensajes(convergecast) &= |V| - 1 = 1 - 1 = 0\\ Tiempo(convergecast) &= Profundidad(T) = 0 \end{aligned}$$

Generalicemos:

Veremos casos donde hay mas de un vertice, pues el anterior ya lo mostramos. En el tiempo cero, los primeros k mensajes enviados son de las k hojas del árbol T que cumplen con la condición enumerada dos en el algoritmo.

Con lo que en el tiempo uno, al menos un proceso recibe r con $1 \leq r \leq k$ mensajes, pues existe el caso que algún vertice tenga k hojas como hijos.

Existen al menos $m \geq 1$ procesos que para el tiempo dos se cumple $HIJOS == recibidos$, pues en caso de ser contrario el algoritmo no terminaria. Si m es uno, llamemos a ese proceso v, entonces v recibio k mensajes.

Sea $A = \{u|\ profundidad(u) == profundidad(T)\}$, entonces en A están las hojas de mayor profundidad. Observese que no necesariamente todos los elementos de A son los que comienzan a mandar mensaje, pues puede existir una hoja de menor profundidad que al ser hoja cumple con la condicion inicial. Sea $B = \{$hojas que no tienen profundidad igual a la profundiad del arbol$\}$.

Si $profundiad(T) == 1$ ya terminamos, pues $|B| = 0$ y para los $|A|$ procesos su padre era la raíz.

Sea $profundiad(T) = d$, así $A = \{u_1, u_2, ..., u_n\}$. Entonces $n+|B|$ procesos envian mensajes en el tiempo cero, no podemos asegurar nada de los $|B|$ mensajes, pero si podemos afirmar que para el tiempo uno los n procesos llenan la capa $d-1$ del árbol, por que para los vértices de esa capa solo es posible que tengan hijos en la profundidad del árbol o sean hojas. Así en el tiempo t podemos asegurar que se llena la capa $d-t$. El algoritmo termina cuando se llena la capa cero. Garantizamos que la capa cero se llena en el tiempo t tal que $d - t = 0$, que es en el tiempo $t = d$.□

La complejidad de los mensajes se sigue de lo antes dicho, cada proceso envía mensaje una única vez, primero los de la capa d y termina hasta que la capa uno se reporta con la raíz.

- El algoritmo de broadConvergecastTree.

Solución:
Suponemos la existencia de un árbol generador, T.

Algoritmo 4 broadConvergecastTree(ID, soyRaiz):

```
1. PADRE, HIJOS, soyRaiz = soyRaiz, vecinos = 0

   Inicialmente hacer
2. Si soyRaiz entonces
3.     send(< START >) a todos en HIJOS

   Al recibir <START> de PADRE:
4. Si |HIJOS| ≠ 0 entonces:
5.     send(< START>) a todos en HIJOS
6. Sino
7.         send(< ok >) a PADRE

   Al recibir <ok> de algún puerto en HIJOS:
8. vecinos++
9. Si vecinos == |HIJOS| entonces:
10.    Si soyRaiz entonces:
11.        Reportar terminación
12.    Sino
13.        send(<ok>) a PADRE
```

Con todos los $START$ se recorre el arbol de raíz hasta las hojas, con lo que se envían $|V|-1$ mensajes $START$. Luego con los ok se recorre el arbol de hojas a raíz con lo que se envían $|V|$ ok. En total se envían $2 \cdot |V| - 1$ mensajes.

Como toma una unidad de tiempo ir de una capa a otra, para cualquier proceso p $START$ le llega en $d(raiz, p)$ unidades de tiempo. Entonces

$$Tiempo(START) = profundidad(T)$$

EL mensaje ok es un poco más complicado, por que es necesario esperar que todos los hijos contesten. Y no siempre ocurre que todos los hijos de un proceso esten a la misma profundidad. Así, en el peor caso donde un vértice espera por al menos uno de sus hijos es cuando: la raíz tiene dos hijos, uno de ellos es hoja y el otro su subarbol tiene profundidad igual a la profundidad del arbol original menos uno. Para que la raíz se reporte va a tener que esperar a que se envien los $START$ que ya sabemos que toma $profundidad(T)$ mas lo que tome el árbol en subir, que tambien es $profundidad(T)$. En este caso la raíz de esperar $profundidad(T)-2$ unidades de tiempo despues de haber recibido el ok de su hijo hoja. Y el peor caso de tiempo en general es cuando ambos hijos tienen subarboles altura $profundidad(T) - 1$. Concluimos:

$$Tiempo(ok) = 3 \cdot profundidad(T)$$

3. ¿Se basan los algoritmos de broadcastTree y convergecast en el conocimiento acerca del número de nodos en el sistema?¿Por qué?

Solución:

Al ser un sistema distribuido un proceso no esta al tanto de los HIJOS de otro proceso y no hay tal cual una varible global que nos indice el número total de vertices, pero si cada proceso no conociera el número de sus hijos convergecast no funcionaría, pues es lo que le permite identificar a un proceso que no esta esperando algún mensaje.

El número de nodos en los arboles resulta ser la suma de los HIJOS de todos los vertices + 1(la raíz), esto por que no hay ciclos.

Por otro lado broadcastTree solo utiliza al PADRE, si tuvieramos un conjunto que indique a los padres no podemos saber mucho acerca del total de procesos en el sistema (arbol no binario), de forma que broadcastTree no se basa en eso.

4. Investiga y explica brevemente el concepto de time-to-live(TTL) usado en redes de computadoras y úsalo para modificar el algoritmo flooding visto en clase, de modo que un lider comunique un mensaje M a los procesos a distancia a lo más d del líder (M y d son entradas del algoritmo); todos los procesos a distancia mayor no deberian recibir M. Da un breve argumento que demuestre que tu algoritmo es correcto, y también haz un analísis de tiempo y número de mensajes.

Solución:

Time to live o tiempo de vida, también conocido como hop limit o limite de saltos, es una herramienta que delimita el lapso de vida de información en una computadora o una red. TTL suele ser implementado con algún contador o con un sello, es decir una secuencia de caracteres (cadenas) o información decodificada, que en cuanto el contador llegue a su objetivo o el sello expire, la información es descartada o revalidad. En las redes de computadoras, TTL previene que un paquete circule indefinidamente. TTL también suele utilizarse en sistemas cache.

Mostramos nuestro algoritmo:

Algoritmo 5 `floodingTTL(ID, Lider, M, d):`

```
1. flag = false

   Inicialmente hacer
2. Si ID == Lider entonces // Soy el lider
3.     flag = true
4.     send(<M, 1>) por todos los puertos

Al recibir algún mensaje <M, cont> de algún puerto:
5. Si not flag and cont < d entonces:
6.         flag = true
7.         send(<M, cont+1>) por todos los puertos
```

Primero pense en crear una variable *cont*, pero los demas procesos no tienen acceso a las variables de los démas, entonces en el caso del lider decidí enviar M junto a 1, que es la distancía de si mismo a que la envia, para el caso donde reciben M y el entero *cont*, sí no habían enviado el mensaje previamente y el entero es menor al rango que todos

reciben entonces el proceso envía el mensaje a todos sus puertos junto al contador incrementado en uno. La condición es menor y no menor o igual por que con el menor garantizo que llegara al sucesor del predecesor de d, es decir d. Y si hubiese sido menor o igual el mensaje llegaría a los distancia $d+1$.

Recordemos que en flooding los mensajes rebotan, si bien los procesos que ya enviaron el mensaje solo lo hacen una vez, estos escuchan el mensaje n veces, donde n es el número de puertos a los que está conectado.

Ahora no podemos garantizar que todo elemento reciba el mensaje, pues es solo para procesos p tales que $d(raiz, p) \leq d$

En la clase habiamos considerado las aristas como bidireccionales de forma que ahora no necesariamente se envian $2|E|$ mensajes. Considere el siguiente conjunto

$$E' = \{uv|\ d(raiz, u) \leq d \wedge d(raiz, v) \leq d\}$$

Así, si seguimos considerando bidireccionales las aristas se envian $2|E'|$ mensajes.

Nota: Para que se envien $2|E|$ aristas se debe cumplir que para cualquier vertice , v, $d(raiz, v) \leq d$.

5. Generaliza el algoritmo convergecast para recolectar toda la información del sistema. Esto es, cuando el algoritmo termine, la raíz debería tener todas las entrdas de todos los procesos. Analiza la complejidad de bits, es decir, el total de bits que son enviados sobre los canales de comunicación. (*hint: Cada mensaje de información puede tomar k bits*).

Solución:

De nuevo suponemos existe el arbol generador, T. Suponiendo que la entrada de los algoritmos es una cadena.

Algoritmo 6 convergecastRecolecta(input, soyRaiz):

```
1. PADRE, HIJOS, soyRaiz = soyRaiz, recibidos = 0, micadena = input

   Inicialmente hacer
2. Si |HIJOS| == 0 entonces
3.     send(< ok, micadena >) a PADRE

   Al recibir <ok, cadena> de algún puerto en HIJOS:
4. recibidos++
5. micadena = micadena++'', ''++cadena
6. Si recibidos == |HIJOS| entonces:
7.     Si soyRaiz entonces:
8.        regresa(micadena) // fin algoritmo
9.     else
10.       send(< ok, micadena >) a PADRE
```

Donde $++$ es la concatenación de cadenas/mensajes.

En el tiempo cero la variable micadena, inicializada con input, vale/contiene k bits para todos los procesos. Los mensajes en el tiempo cero siguen partiendo de las hojas.

Si tenemos un único proceso, en el tiempo cero acaba y el total de bits enviados es cero.

Si $profundidad(T) == 1$ entonces los $|V|-1$ vertices envian en el tiempo cero su entrada de k bits, con lo que el total de bits enviados es $(|V|-1)\cdot k$. Para el tiempo uno raíz concatena su cadena inicial con las $|V|-1$ y el algoritmo termina. Si consideramos el reporte final de la raíz circulan $(|V|)\cdot k$ bits.

Supongamos $profundidad(T) = d$.

Para cada mensaje que le llegue a cualquier proceso, este proceso concatena la cadena recibida con la cadena micadena, inicializada en tiempo cero con la entrada del proceso. Así en el tiempo uno cuando sea el turno de enviar la variable micadena para todo proceso en la capa $d-1$, esta pesara $(|HIJOS|+1)\cdot k$, de manera que en el tiempo $t+1$, se garantiza que todo proceso en la capa $d-t$ envió su micadena a su padre que pesa $(|HIJOS|+1)\cdot k$ más lo acumulado por cada hijo.

Entonces el total de bits enviados sobre los canales de comunicación, si consideramos el reporte final de la raíz, es la suma de todos los vertices de las capas c_i con $i \in [0, 1, ..., d]$, que por definición es $|V|$. Si se considera que el algoritmo termina en cuanto la capa 1 le reporta a la raíz entonces son los vertices en las capas $[1, ..., d]$, que es $(|V|-1)\cdot k$ bits.

6. *a*) Da un algoritmo distribuido para contar el número de vértices en un árbol enraízado T, iniciando en la raíz.
Solución:

Algoritmo 7 broadConvergecastTreeCuentaV(ID, soyRaiz):

```
1. PADRE, HIJOS, soyRaiz = soyRaiz, vecinos = 0, vertices = 1

  Inicialmente hacer
2. Si soyRaiz entonces
3.     Si |HIJOS == 0| entonces:
4.        regresa 1
5.     Sino
6.        send(< START >) a todos en HIJOS

  Al recibir <START> de  PADRE:
7. Si |HIJOS| ≠ 0 entonces:
8.     send(< START>) a todos en HIJOS
9. Sino
10.       send(< ok, 1 >) a PADRE

  Al recibir <ok, accum> de algún puerto en HIJOS:
11. vecinos++
12. vertices = vertices + accum
13. Si vecinos == |HIJOS| entonces:
14.    Si soyRaiz entonces:
15.       Regresa vertices // fin
16.    Sino
17.       send(<ok, vertices>) a PADRE
```

---

b) Extiende tu algoritmo para una gráfica arbitraria G.

Solución:

Algoritmo 8 `broadConvergecastGraphCuentaV(ID, soyRaiz):`

Ejecutar inicialmente:
`1. buildSpanningTree(ID, root, <hola?>)`
Dado el árbol T obtenido en el paso uno:
`2. broadConvergecastTreeCuentaV(ID, soyRaiz) // Algoritmo 7 de esta tarea.`

---

7. Da un algoritmo distribuido para contar el número de vértices en cada capa de un árbol enraízado T de forma separada. Analiza la complejidad de tiempo y la complejidad de mensajes de tu algoritmo.

Solución:

Se nos ocurre la siguiente forma de ver el número de elementos por capas:

$$\texttt{capa } 0 = Raiz$$
$$\texttt{capa } 1 = |Raiz.HIJOS|$$
$$\texttt{capa } 2 = \sum_{v \in V_1} |v.HIJOS| \qquad \texttt{Donde } V_1 = \{v \in V | d(raiz, v) = 1\}$$
$$\vdots$$
$$\texttt{capa } n = \sum_{v \in V_{n-1}} |v.HIJOS| \qquad \texttt{Donde } V_{n-1} = \{v \in V | d(raiz, v) = profundidad(T) - 1\}$$

De forma que para obtener los vertices de una capa k, los elementos en la capa $k-1$ tienen que regresar la cardinalidad de sus HIJOS.

Para nuestro algoritmo cualquier procesos, p, debe tener la siguiente memoria/composición:

$u\{$
$padre =$ `Puerto`
$soyRaiz =$ `Booleano`
$HIJOS =$ `Conjunto de puertos`
$profundidad = n \in \mathbb{N}$
$\}$

Primero es necesario hacer mención de un algoritmo auxiliar que a cada proceso llena la variable profundidad con su profundidad correspondiente y que regresa la profundidad del árbol.

Algoritmo 8 `profundidadT(soyRaiz):`

```
1. PADRE, HIJOS, profundidad, soyRaiz = soyRaiz, vecinos = 0, max = 0

  Inicialmente hacer
2. Si soyRaiz entonces
3.     profundidad = 0
4.     Si |HIJOS| == 0 entonces:
5.        regresa 0
6.     Sino
7.        send(< 1 >) a todos en HIJOS

  Al recibir < n > de  PADRE:
8. profundidad = n
9. Si |HIJOS| == 0 entonces:
10.     send(< termine, n>) a PADRE
11. Sino
12.       send(< n+1 >) a todos en HIJOS

  Al recibir <termine , n> de algún puerto en HIJOS:
13. vecinos++
14. Si n >max entonces:
15.     max = n
16. Si vecinos == |HIJOS| entonces:
17.   Si soyRaiz entonces:
18.       Regresa max // fin
19.    Sino
20.       send(<termine, max>) a PADRE
```

El algoritmo comienza con la raíz indicando su profundidad en cero e indicandole a todos sus vecinos que su profundidad es uno. En caso de que la raíz no tenga vecinos, la profunidad del árbol es cero.

Cuando cualquier vertice recibe un unico entero, el proceso inicializa su variable profundidad al entero recibido. Si es hoja, entonces ya termino y le notifica al padre que termino y su altura, si no le indica cual es su profundidad a todos en HIJOS.

Al recibir $< termine, n >$, el proceso determina si la n recibida es mayor a su variable local max si sí ocurre entonces fija $max = n$. Si el número de $< termine, n >$ recibidos es igual al número de HIJOS, ya terminó. Si se trata de la raíz este regresa su variable max, si no le comunica a su padre que termino y la profunidad maxima obtenida de sus subarboles.

El algoritmo recorre el arbol de arriba a abajo para poner las profundidades, lo que toma en tiempo $profundidad(T)$ y luego lo recorre de abajo a arriba para obtener la profundidad máxima lo que de nuevo toma $profundidad(T)$. Entonces este algoritmo auxiliar toma $2 \cdot profundidad(T)$ en tiempo.

Cada arista envia un mensaje para poner las profundidades y otro para obtener la profundidad maxima, entonces el algoritmo envía $2|E|$ mensajes.

Así, ahora, cada proceso conoce su profundidad y podemos aplicar la idea mencionada al inicio.

Algoritmo 9 `getCapas(soyRaiz)`:

```
1. profArbol = profundidadT(soyRaiz), totalcapa = 0, vecinos = 0
2. PADRE, HIJOS, profundidad, soyRaiz = soyRaiz, cadena = ""

  Inicialmente hacer
3. Si soyRaiz entonces
4.     cadena = "1"
5.     Si |HIJOS| == 0 entonces:
6.        regresa "1"
7.     Sino
8.        cadena = cadena + +"," + +|HIJOS|
9.        send(< :dame, 1 > // Le pide a todos los vertices de la capa uno envien |HIJOS|


  Al recibir <:dame, capa> de PADRE:
10. Si profundidad == capa entonces:
11.     send(< |HIJOS|, capa >) a PADRE
12. Sino
13.     send(< :dame, capa >) a todos en HIJOS

  Al recibir < hijos, capa > de algún puerto en HIJOS:
14. vecinos += 1, totalcapa += hijos
15. Si vecinos == |HIJOS| entonces:
16.   Si soyRaiz entonces:
17.       cadena = cadena + +"," + +hijos
18.       Si capa == profArbol − 1 entonces:
19.          Regresa cadena // fin algoritmo
20.       Si no
21.         send(<: dame, capa + 1 >) a HIJOS
22.    Si no
23.       send(< totalcapa, capa >) a PADRE
```

Es facíl ver que el algoritmo va de arriba a abajo, puede que no sea el algoritmo más optimo, pero no supe manejar mi tiempo y no hay chance de pensar en uno mejor. Al final el algoritmo regresa la cadena $n_0, n_1, ..., n_{profundidad(T)}$ donde cada n_i representa el número de vertices por capa.

El algoritmo recorre desde la raíz hasta cada una de las capas y de regreso, con excepción de la de mayor profundida, de forma que en tiempo toma $profundidad(t) * profundidad(T)$.

Por otro lado el algoritmo manda mensajes hasta la capa que le importa $n = profundidad(T) - 1$ veces, y el mismo numero de veces para arriba, cuando reportan el número de HIJOS para los procesos en una capa menor a la solicitada.