UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 2

INTEGRANTES

Torres Valencia Kevin Jair - 318331818
Aguilera Moreno Adrián - 421005200
Natalia Abigail Pérez Romero - 31814426

PROFESOR

Miguel Ángel Piña Avelino

AYUDANTE

Pablo Gerardo González López

ASIGNATURA

Computación Distribuida

27 de septiembre de 2022

1. Investiga y explica brevemente el concepto de time-to-live (TTL) usado en redes de computadoras, y úsalo para modificar el algoritmo de flooding visto en clase, de modo que un líder 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 deberán recibir m. Da un breve argumento que demuestre que tu algoritmo es correcto, y también haz un análisis de tiempo y número de mensajes.

Time-to-live (TTL) se refiere al tiempo o "saltos" que un paquete tiene que realizar para existir en una red antes de que sea descartado por un router. Y funciona cuando un paquete de información es creado y enviado por internet, existe un riesgo de que este continúe pasando de un router a otro de manera indefinida. Para evitar esta situación, los paquetes están diseñados con una expiración llamada time-to-live o limite de saltos.

Recordando el algoritmo de flooding visto en clase es:

Algorithm 1 flooding(ID,Lider,M)

```
flag = False
Ejecutar inicialmente:
if ID = Lider then
    flag = True
    send (< M >) por todos los puertos
end if
Al recibir < M > por algún puerto:
if not flag then
    flag = True
    send (< M >) por todos los puertos
end if
```

Se propone que primero se escoja el lider y al hacerlo se envié el $(< M, 1 >)$, indicando su distancia (la cuál es 1). Por otro lado, cuando se recibe $< M, count >$ y no han enviado su mensaje, y el entero es menor a todos los demás recibidos, implica que se envié el mensaje a todos los puertos con su contador, el cuál irá incrementando en uno. La condición (la segunda), tiene que ser menor para asegurar que se envié a 'uno antes' de su predecesor D.

Algorithm 2 TTL(ID,Lider,M,D)

```
flag = False
Ejecutar inicialmente:
if ID = Lider then
    flag = True
    send (< M, 1 >) por todos los puertos                ▷ 1 es su distancia
end if
Al recibir < M, count > por algún puerto:                ▷ count es un contador
if not flag & count < D then
    flag = True
    send (< M, count + + >) por todos los puertos        ▷ count incrementa en uno
end if
```

-Análisis de tiempo:
Como todo proceso recibe el mensaje M en a lo más tiempo $D + 1$, en el mejor de los casos es D.
-Análisis de mensajes:
Como cada proceso envía solo una copia de M a sus vecinos, así que cada arista transporta a lo más una copia de M por mensaje. Por el algoritmo original, observamos que M se envía a lo más 2 veces por arista, por lo que la cantidad de mensajes es a lo más $2|E|$.

2. Considera un sistema distribuido representado como una gráfica de tipo anillo, cuyos canales son bidireccionales, con n = mk procesos, con $m > 1$ y k es impar. Los procesos en las posiciones $0,\ k,\ 2k, ..., (m-1)k$ son marcados 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 líderes 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 el algoritmo con $k = 5$ y $m = 4$.

Pseudocódigo 1: Algoritmo reclutamiento

```
1  Algoritmo reclutamiento(id, soyLider):
2
3  Inicialmente hacer:
4    if soyLider then
5      parent = id
6      send(<recluta>) a ambos vecinos
7    else
8      parent = ⊥
9
10 Al recibir <recluta> desde p hacer:
11   if parent = ⊥ then
12     parent = p
13     send(<recluta>) a mi vecino que no es p
```

▷ Empecemos dando respuesta a la pregunta ¿Cuál es el tamaño mínimo y máximo posible de un árbol?. Para contestar esta pregunta análicemos varias cosas:

1. Sabemos que la cantidad de procesos es múltiplo de la cantidad de líderes, pues si

$$n, m, k \in \mathbb{N} \Rightarrow \frac{n}{k} = m \in \mathbb{N}.$$

   Entonces, la cantidad de líderes (k) es proporcional a la cantidad de seguidores (m), hasta el momento no sabemos si a cada líder le corresponda la misma cantidad de seguidores basados en el algoritmo de `reclutamiento`.

2. Recordemos que k es impar y por tanto tiene la forma $k = 2r + 1$ ($r \in \mathbb{N} \cup \{0\}$). Ahora, nos podemos preguntar ¿cada cuántos procesos hay un líder contando desde el último anterior o próximo? la respuesta es cada k procesos y la diferencia entre estos es de $k - 1$ procesos, pues los líderes están ubicados en múltiplos de k, luego

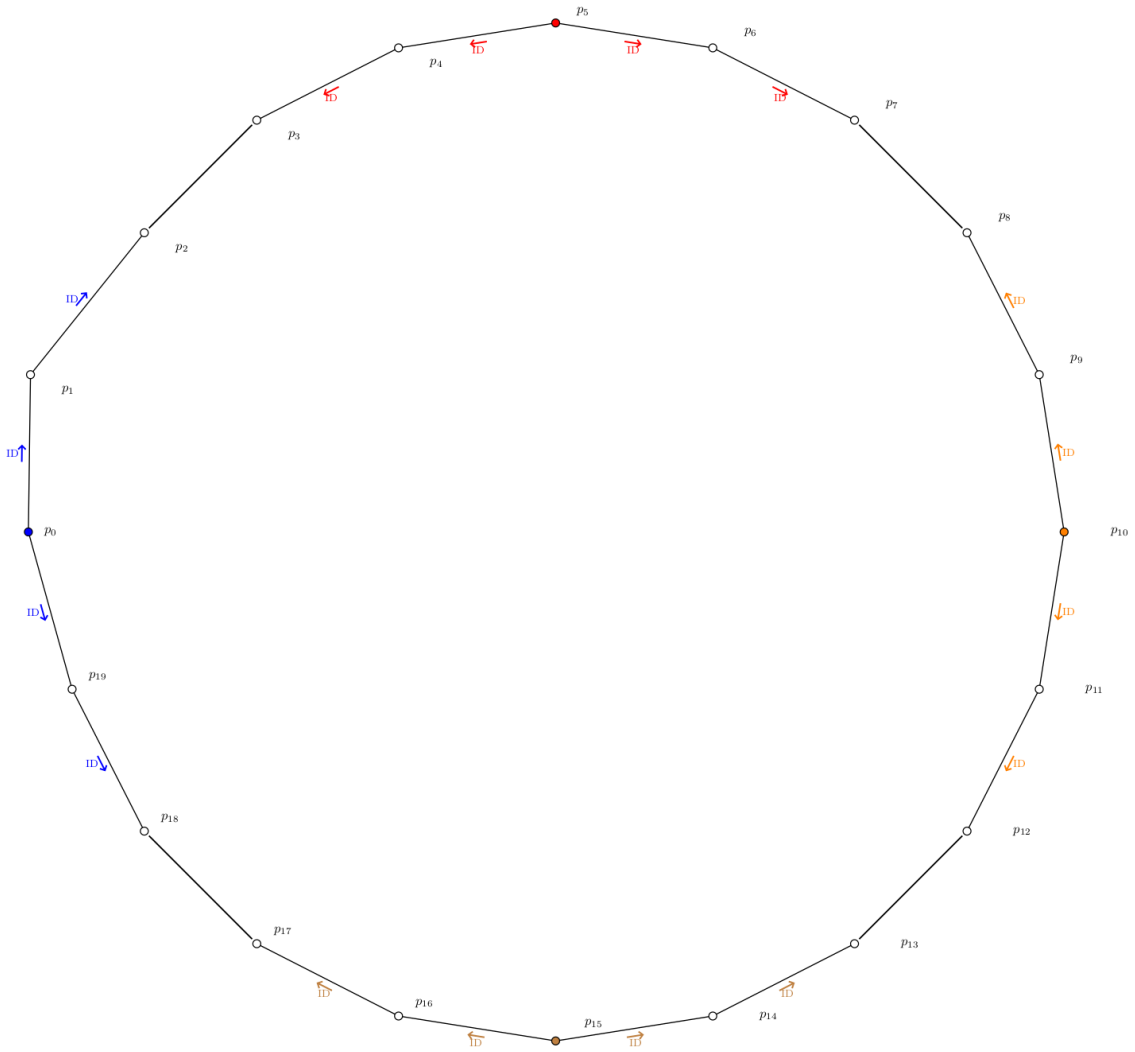
$$k - 1 = (2r + 1) - 1 = 2r.$$

   Inicialmente, los procesos líderes empiezan a reclutar (línea 6). Después de la primer ronda los procesos reclutados reclutan a sus vecinos que no son su padre (línea 13). Eventualmente todo los procesos seguidores son relcutados, como todo esto (en cada proceso reclutado lo que sigue es reclutar a su vecino) pasa al mismo tiempo (en cada ronda los procesos líderes tienen la misma cantidad de descendientes) podemos notar que la diferencia de procesos entre cualesquiera dos procesos líderes es dividida entre 2 para que cada líder termine por ser ancestro común de exactamente la mitad, adyacente al líder en cuestión, de esos procesos.

   Hasta este momento sabemos que un líder llegará a tener r descendientes por lado, en total cada líder tendrá $2r$ descendientes.

3. Como los árboles están enraizados por líderes y cada líder tiene $2r$ descendientes. Entonces, el total de nodos en los arboles generados será $2r + 1 = k$. Concluimos que el tamaño por árbol es de k sin importar como es m o k.

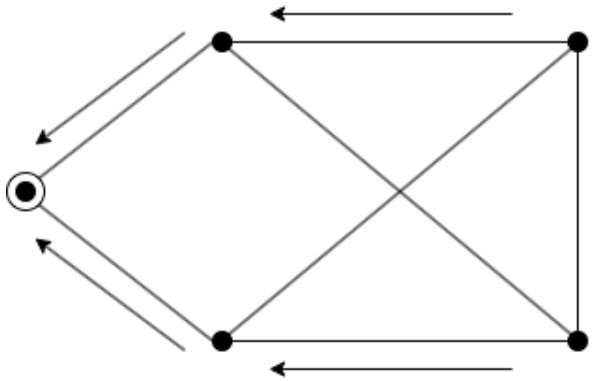
Por último, veamos como se ve la gráfica después de haber ejecutado el algoritmo de `reclutamiento`:



Los árboles en este caso se pueden formar desde cada nodo representado por algún color[I] y hasta donde se indica con las flechas del respectivo color. ◁

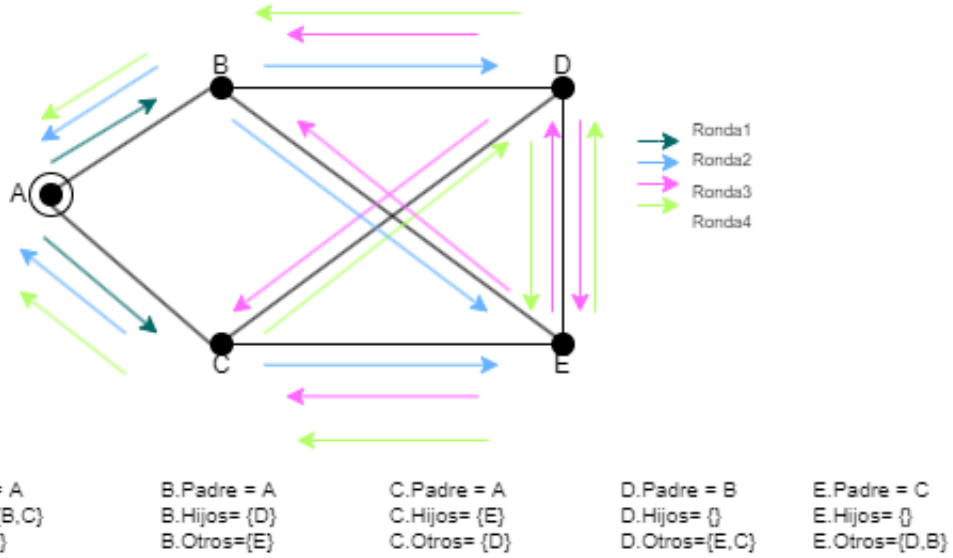
[I] estos son los nodos distinguidos como líderes.

3. ¿El árbol generador de la figura 1 puede obtenerse en alguna ejecución del algoritmo BFS visto en clase? de ser el caso, describe la ejecución, y de no serlo, explica por qué no se puede. Haz lo mismo con el algoritmo DFS.



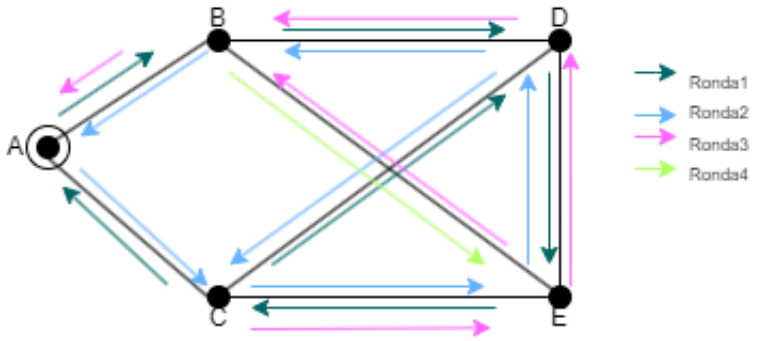
Para poder visualizar mejor el árbol se le asigno un identificador a cada uno de sus nodos (A, B, C, D, E)

A) Caso BFS:



Si, bueno como se comienza desde nuestro caso A, entonces se declara como padre y este le manda $< BFS, A >$ tanto a B, como C, de ahi tanto B y C manda $< BFS, ID >$ a todos sus vecinos, y al mismo tiempo manda a A, definiendolo como su padre. Y asi hasta llegar a nuestro vértice D el cuál le manda a nuestro vértice E. Ya que todos los vértices hayan recibido su respectivo mensaje con $< BFS, ID >$, devolverán un mensaje a su "padre"por decirlo así con $< alredy >$. De los vértices que reciban ya sea $< alredy >$ o $< parent >$ se tendrán que cumplir con la codición dada el algoritmo BFS.

B) Caso DFS:



A.Padre = A	B.Padre = A	C.Padre = E	D.Padre = B	E.Padre = D
A.Hijos= {B}	B.Hijos= {D}	C.Hijos= {A,D}	D.Hijos= {E}	E.Hijos= {B,C}
A.SinExpl={}	B.SinExpl={}	C.SinExpl= {}	D.SinExpl={}	E.SinExpl={}

Al comenzar desde A, se puede obesrvar que solo manda un mensaje a uno sus posibles vecinos. Por lo que, en cierta forma obliga a decidir entre varios posibles caminos para distribuir el mensaje, pero no se puede lograr, generar tal árbol, que se nos pide, ya que al ser una gráfica con vértices impares, debe haber habido otro vértice, se pudo haber enviado un mensaje más y así lograr el árbol objetivo.

4. Describe un algoritmo distribuido para construir un árbol generador sobre una gráfica arbitraria G, utilizando el algoritmo de elección de líder eligeLider para determinar la raíz del árbol y también el algoritmo BFS. Analiza la complejidad de tiempo y de mensajes.

Algorithm 3 arbolGeneradorConBFS(ID,total)

$PADRE = \perp, HIJOS = \emptyset, OTROS = \emptyset$
$LIDER = \perp$
Si no he recibido algún mensaje y no he decidido el lider
if $LIDER == \perp \ \ and \ \ PADRE == \perp$ **then** ▷ Algorimo eligeLider
    $LIDER = ID, ronda = 0$
    Ejecutar en todo momento $t \geq 0$
    $send < LIDER >$ a todos los vecinos
    Al recibir mensaje de todos los vecinos en tiempo $t \geq 1$
    $mensajes = \{< L_1 >, < L_2 >, ... < L_d >\} \cup LIDER$
    $LIDER = Max(mensajes)$
    $ronda = ronda + 1$
    if ronda == total **then** ▷ Termina elección de lider
        if $LIDER \ \ and \ \ PADRE == \perp$ **then** ▷ Algoritmo BFS
            $send < BFS, ID >$ a todos mis vecinos
            PADRE = ID
        end if
        Al recibir $< BFS, j >$ desde el vecino p_j:
        if $PADRE == \perp$ **then**
            $Padre = j$
            $send(< parent >)$ a p_j
            $send(< BFS, ID >)$ a todos excepto p_j
        else
            $send(< already >)$ a p_j
        end if
        Al recibir $< parent >$ desde el vecino p_j
        $HIJOS = HIJOS \cup \{p_j\}$
        if $HIJOS \cup OTROS == VECINOS - PADRE$ **then**
            Terminar
        end if
        Al recibir $< already >$ dede p_j
        $OTROS = OTROS \cup P_j$
        if $HIJOS \cup OTROS == VECINOS - PADRE$ **then**
            Terminar
        end if
    end if
end if

Analiza la complejidad de tiempo y de mensajes.

Complejidad de tiempo

Este algoritmo utiliza dos algoritmos:

1. Primero eligeLider para elegir un lider, el cual utiliza una variable para contar el número de rondas que depende de *total*, ya que al final se tiene $total = ronda$ por lo tanto su complejidad en tiempo es $total = n$

2. En la segunda parte se utiliza el algoritmo BFS usando el LIDER que se eligio antes.

   Al principio el LIDER le envia el mensaje $< BFS, ID >$ a todos su vecinos, los cuales pueden recibir:

   a) $< BFS, ID >$, y si no tiene PADRE le envía un mensaje a sus vecinos y al nodo del que recibio el mensaje para convertirlo en su PADRE. Si tiene PADRE le envia un mensaje al nodo del que recibio el mensaje para avisarle.

b) $< parent >$, verifica si ya ha visitado sus vecinos, si lo hizo termina el algoritmo.

c) $< already >$, verifica si ya ha visitado sus vecinos, si lo hizo termina el algoritmo.

Los casos descritos anteriormente se evaluan para cada nodo en n en una ronda. Podemos pensar que el peor de los casos se tendrá que la gráfica es un camino y el LIDER está en el centro de manera que tarda la máxima distancia entre el LIDER y cualquier vértice de la gráfica, es decir, su complejidad es $O(diam(G))$

Por lo tanto la complejidad de tiempo de todo el algoritmo es $O(n + diam(G))$
Complejidad de mensajes
Este algoritmo utiliza dos algoritmos:

1. Primero eligeLider para elegir un lider, el cual por cada nodo $total = n$ envia un mensaje $< LIDER >$ a todos sus vecinos, por tanto su complejidad es $sum_v n * deg(v) = O(n|E|)$
2. En la segunda parte se utiliza el algoritmo BFS usando el LIDER que se eligio antes. De manera similar cada nodo $total = n$ envia el mensaje $< BFS, ID >$ a sus vecinos con la diferencia de que si ya tiene un padre se envia el mensaje, por tanto su complejidad es $\sum_v n * deg(v) = O(n|E|)$

Por lo tanto la complejidad de mensajes de todo el algoritmo es $O(2n|E|)$

5. El algoritmo puede mejorar su complejidad de tiempo si se ejecutan de forma paralela los dos algoritmos anteriores, es decir, si se ejecuta la elección de líder y la construcción del árbol BFS. Da un algoritmo distribuido que realice esto y muestra que es correcto. Adicionalmente, compara la complejidad de tiempo respecto al algoritmo anterior.

Algorithm 4 arbolGenerador (ID,total)

1: $PADRE = \perp, \quad HIJOS = \emptyset, \quad OTROS = \emptyset$
2: Si no he recibido algún mensaje
3: **if** $PADRE == \perp$ **then**
4: $\quad LIDER = ID$
5: $\quad send(< BFS, ID, LIDER >)$ a todos mis vecinos
6: $\quad PADRE = ID$
7: **end if**
8:
9: Al recibir $< BFS, ID, LIDER >$ desde el vecino p_j
10: **if** $PADRE == \perp$ **then**
11: $\quad PADRE = j$
12: $\quad mensajes = \{< L_1 >, < L_2 >, \ldots, < L_d >\} \cup LIDER$
13: $\quad LIDER = max(mensajes)$
14: $\quad send(< parent, LIDER >)$ a p_j
15: $\quad send(< BFS, ID, LIDER >)$ a todos excepto p_j
16: **else**
17: $\quad send(< already >)$ a p_j
18: **end if**
19:
20: Al recibir $< parent, LIDER >$ dede el vecino p_j
21: $HIJOS = HIJOS \cup \{p_j\}$
22: $mensajes = \{< L_1 >, < L_2 >, \ldots, < L_d >\} \cup LIDER$
23: $LIDER = max(mensajes)$
24: $send(< LIDER >)$ a todos en HIJOS y a PADRE
25:
26: Al recibir $< already >$ desde p_j
27: $OTROS = OTROS \cup \{p_j\}$
28:
29: Al recibir $< LIDER >$ desde p_j
30: **if** $LIDER ==< LIDER > \quad and \quad HIJOS \cup OTROS == VECINOS - PADRE$ **then**
31: $\quad$ **Terminar**
32: **else**
33: $\quad LIDER =< LIDER >$
34: $\quad send(< LIDER >)$ a HIJOS y PADRE
35: **end if**

Podemos describir la ejecución del algoritmo como sigue: Al principio de la ejecución cualquier nodo que se tome al principio será el lider y la raíz del árbol BFS. Luego le envia $< BFS, ID, LIDER >$ a todos sus vecinos, los cuales si no tienen PADRE asigna p_j como su padre y determinan el LIDER (el maximo de los mensajes $< LIDER >$ que ha recibido), lo asigna como su LIDER, le envia a p_j un mensaje que lo asigna como su padre y le dice el nuevo LIDER, cuando el PADRE recibe el mensaje añade el nodo al conjunto de sus hijos, calcula nuevamente el LIDER y se lo envia a sus HIJOS y a su PADRE. Los cuales registran su nuevo LIDER si este es diferente al nodo que tenian almacenado. Si el nodo ya tiene PADRE entonces envian el mensaje $< already >$ al nodo p_j, el cual inserta este al conjunto de sus hijos.
El último segmento lineas 29-35 termina cuando el nodo ya visito a todos sus vecinos (o no tiene hijos) y el LIDER que recibe es igual al que tiene.
Afirmación: El algoritmo `arbolGenerador` es correcto, es decir, cumple con las propiedades de validez y acuerdo. Además construye un árbol con raiz.
Acuerdo. Todos los procesos acuerdan un mismo valor.

Al terminar cualesquiera 2 procesos p_i y p_j con variables $LIDER_i$ y $LIDER_j$, se cumple que $LIDER_i = LIDER_j$ Observemos que para todo tiempo $d > 0$, todos los procesos que están a distancia a lo más d del proceso con el ID máximo, tiene ese ID en la variable LIDER.
Por inducción sobre d:
Caso base: $d = 0$. Es claro que $LIDER = ID$ para el proceso con el ID máximo.
Hipótesis de inducción: Para todo proceso a distancia $d - 1$ del proceso con ID máximo tiene dicho ID en su variable LIDER.
Paso inductivo: Consideremos un proceso p_i a distancia d del proceso con el ID máximo. A partir de la H.I, sabemos que existe un proceso p_j a distancia $d-1$ del proceso con ID máximo con $LIDER = ID.p_j$ vecino de P_i
Por la ejecución del algoritmo sabemos $LIDER = ID.p_j$ cuando al recibir $< LIDER >$ sucede que

$$LIDER == < LIDER > \quad and \quad HIJOS \cup OTROS == VECINOS - PADRE$$

y tenemos dos casos p_j recibio el mensaje de su padre o de sus hijos, si lo recibio de su padre en este caso p_i tiene el $LIDER$ =ID máximo. Si lo recibio de sus hijos entonces tenemos otros dos casos el ID de p_j era el ID máximo o no lo era; en caso de que no lo fuera envia el nuevo ID máximo a su padre y a sus hijos, quienes caen nuevamente en este caso. Al final de la ejecución $LIDER == < LIDER >$ es decir el ID máximo que recibe del PADRE y de los HIJOS es el mismo.
Validez. Al terminar el algoritmo todos los procesos tienen como LIDER un ID que fue entrada de algún proceso.
Al terminar todo proceso tiene como LIDER un ID que fue entrada de algún proceso. Esto es fácil de observar, ya que $LIDER = max(mensajes)$ y esto es construido a partir de las propuestas de los vecinos.

Podemos garantizar que la gráfica que regresa `arbolGenerador` es un árbol con raíz por los argumentos que se usan en BFS: 1. Todo nodo es alcanzable desde la raíz si el sistema es conexo. Dado que a todo nodo se le asigna un padre es una contradicción suponer que no es alcanzable. 2. No tiene ciclo. Si existiera un ciclo entonces un proceso tendría dos padres pero esto es una contradición porque un nodo solo tiene un padre por la linea 11.
Compara la complejidad de tiempo respecto al algoritmo anterior.
En este caso el algortimo hace a BFS al mismo tiempo que elige un lider por lo que tarda la máxima distancia entre el LIDER y cualquier vértice de la gráfica, es decir, su complejidad de tiempo es $O(diam(G))$

6. Prueba la siguiente afirmación:
El algoritmo BFS construye un árbol enraízado sobre un sistema distribuido con m aristas y diámetro D, con complejidad de mensajes $O(m)$ y complejidad de tiempo $O(D)$.

$\triangleright$ Para este problema damos por hecho que `BFS` construye un árbol enraízado de diámetro D (esto por la afirmación 9 y 11).
Ahora, analicemos las complejidades de este algoritmo, para esto observemos dos posibles casos:

1. La complejidad de tiempo es $O(D)$.

   Sea G una gráfica conexa de diametro D a la cuál se le aplica `BFS` y nos genera T con algún vértice distinguido v como líder.

   Por las líneas 7 - 9, el líder v se convierte en un nodo enraízado para T. Ahora, ¿cuál es el número de rondas antes de que nuestro algoritmo termine para todos los procesos en G?, de acuerdo al algoritmo los descendientes de v en T serán el resto de los nodos.

   El algoritmo, por proceso p_i, termina cuando `Hijos` $\cup$ `Otros` contiene a todos los vecinos del respectivo p_i, a excepción de su padre (líneas 21 y 26). Así, una vez que `BFS` inicia en v los p_i restantes no hacen nada (`BFS` ha iniciado pero no tiene instrucciones de distribuirse aún), en cuanto los vecinos de p_i empiezan a realizar sus tareas, empiezan a propagar su ID para convertirse en padre, si es que se cumple la condición necesaria.

   Pensemos en la trayectoria t más larga que inicia en v y termina en el último proceso, p, en terminar. Esta trayectoria tiene longitud equivalente al diámetro de G, pues sino fuese así pasaría que

   a) La longitud de t es menor que `diam(G)`. En este caso, existe (por definición de diámetro) algún proceso que al conectarse con t por alguna arista e y que tiene como vecino a p, permite que t siga siendo trayectoria y de hecho $t - e$ tiene longitud 1 mayor que t, lo que contradice que t fuera la trayectoria más larga.

   b) La longitud de t es mayor que `diam(G)`. En este caso, t debe contener al menos un ciclo por lo que t no sería trayectoria y esto es contradictorio.

   En conclusión t es de longitud igual que el diámetro de G. En el momento en que p recibe `parent` desde alguno de sus puertos, ya el resto nodos han recibido `parent` o lo están recibiendo de manera simultánea. Esto pasa en la ronda $D - 1$, pues es en la ronda 0 donde inicia a realizar sus tareas al menos un proceso (v). En la siguiente ronda envia realiza sus tareas y recibe sus respectivos mensajes. Si no terninara en esta ronda, entonces hay un proceso que no tiene padre y p no es último!!, esto es contradictorio. Como p termina en la ronda siguiente a la que le llega padre, podemos notar que este último proceso termina en tiempo

   $$(D - 1) + 1 = D \in O(D)$$

2. La complejidad de mensajes es $O(m)$.

   Sabemos que cada proceso envía sus respectivos mensajes a sus vecinos, exceptuando a su padre. Eventualmente los procesos ocupan como puerto de salida a todas las aristas que conecten a sus vecinos, como la gráfica es conexa, entonces al finalizar el algoritmo se utilizan todas las aristas.

   En el peor de los casos, un proceso que recibe `parent` ya tiene asignado a un padre, entonces debe regresarle `already` por el mismo puerto. La complejidad de mensajes será, entonces

   $$m + c \cdot m \in O(m)$$

   donde c nos indica la cantidad de procesos que caén en este peor caso, como sabemos que esto no pasa cuando se recibe `parent`, el número de peores caso no es m.

$\triangleleft$