**DEFINICIÓN 1: Difusión (Broadcast)**

Una operación de difusión la inicia un único proceso, la fuente. La fuente quiere enviar un mensaje a todos los demás nodos del sistema.

**DEFINICIÓN 2: Distancia, radio, diámetro**

La distancia entre dos nodos $u$ y $v$ en una gráfica no dirigida $G$ es el número de saltos de un camino mínimo entre $u$ y $v$. El radio de un nodo $u$ es la distancia máxima entre $u$ y cualquier otro nodo de la gráfica. El radio de una gráfica es el radio mínimo de cualquier nodo de la gráfica. El diámetro de una gráfica es la distancia máxima entre dos nodos arbitrarios.

**OBSERVACIÓN.** Es evidente que existe una estrecha relación entre el radio $R$ y el diámetro $D$ de una gráfica, como $R \leq D \leq 2R$.

**TEOREMA 1: Límite inferior de difusión**

La complejidad del mensaje de difusión es al menos $n - 1$. El origen del radio es un límite inferior para la complejidad en tiempo.

*Demostración.* Todos los nodos deben recibir el mensaje. □

**OBSERVACIÓN.** Se puede utilizar un árbol generador precalculado para realizar la difusión con una complejidad de mensajes ajustada. Si el árbol generador es un árbol generador de búsqueda amplia 'BFS' (para un origen determinado), la complejidad en tiempo también se reduce.

**DEFINICIÓN 3: Gráfica limpia**

Una gráfica (red) es limpia si los nodos no conocen la topología de la gráfica.

**TEOREMA 2: Límite inferior de difusión limpia**

Para una red limpia, el número de aristas es un límite inferior para la complejidad de los mensajes de difusión.

*Demostración.* Si no prueba todas las aristas, podría perderse toda una parte de la gráfica. □

**TEOREMA 3**

Cada proceso recibe $M$ tras un tiempo máximo de $D$ y un máximo de $|E|$ mensajes, donde $D$ es el diámetro de la red y $E$ es el conjunto de aristas (dirigidas) de la red.

*Demostración.* Complejidad de los mensajes: Cada proceso sólo envía $M$ a sus vecinos una vez, por lo que cada arista transporta como máximo una copia de $M$.
Complejidad en tiempo: Por inducción sobre $d(raiz, v)$, demostraremos que cada $v$ recibe $M$ por primera vez a más tardar en el tiempo $d(raiz, v) \leq D$. El caso base es cuando $v = raiz$, $d(raiz, v) = 0$; aquí la raíz recibe el mensaje en el tiempo 0. Para el paso inductivo, Sea $d(raiz, v) = k, k > 0$. Entonces $v$ tiene un vecino $u$ tal que $d(raiz, u) = k - 1$. Por la hipótesis de inducción, $u$

recibe $M$ por primera vez a más tardar en el tiempo $k-1$. A partir del código, $u$ envía entonces $M$ a todos sus vecinos, incluido $v$; $M$ llega a $v$ a más tardar en el tiempo $(k-1)+1=k$. □

**OBSERVACIÓN.** La demostración de la complejidad en tiempo también demuestra que es correcto: cada proceso recibe $M$ al menos una vez.

**Algorithm 1** Broadcast Ingenuo - Código para el proceso $p_i$

```
1: neighbors_i = { Conjunto de vecinos de p_i }
2: seen_message = false
3: if p_s = p_i then
4:     seen_message = true
5:     send M to all neighbors
6: end if

7: when M is received from p_j do
8: begin:
9:     if seen_message = false then
10:        seen_message = true
11:        send M to all neighbors
12:    end if
13: end
```

**Algorithm 2** Broadcast - Código para el proceso $p_i$

```
1: Initially do
2: begin:
3:     if p_s = p_i then
4:         data = mensaje que se quiere difundir
5:         for each j ∈ children_i do
6:             send GO(data) to p_j
7:     else
8:         data = 0
9:     end if
10: end

11: when GO(data) is received from p_j do
12: begin:
13:    for each k ∈ children_i do
14:        send GO(data) to p_k
15:    end for
16: end
```

**OBSERVACIÓN.** Si el nodo $v$ recibe el mensaje primero del nodo $u$, entonces el nodo $v$ llama padre al nodo $u$. Esta relación de parentesco define un árbol generador $T$. Si el algoritmo de inundación

(broadcast ingenuo) se ejecuta en un sistema síncrono, $T$ es un árbol generador de búsqueda amplia (BFS con respecto a la raíz).

**OBSERVACIÓN.** En sistemas asíncronos el algoritmo de inundación (broadcast ingenuo) termina después de $R$ unidades de tiempo, siendo $R$ el radio de la fuente. Sin embargo, es posible que el árbol generador construido no sea un árbol generador de búsqueda amplia (BFS).

**DEFINICIÓN 4: Convergencia (Convergecast)**

Una operación de convergencia la inicia todo proceso que se considere hoja. Los procesos del sistema envían un mensaje a la fuente o raíz.

**OBSERVACIÓN.** Convergecast es lo mismo que broadcast, pero al revés: en lugar de que una raíz envíe un mensaje a todos los demás nodos, todos los demás nodos envían información a una raíz.

**Algorithm 3** Convergecast - Código para el proceso $p_i$

1: Initially do
2: **begin:**
3: $v_i$
4: **if** $children_i = 0$ **then**
5: send BACK($(i, v_i)$) to $parent_i$
6: **end if**
7: **end**

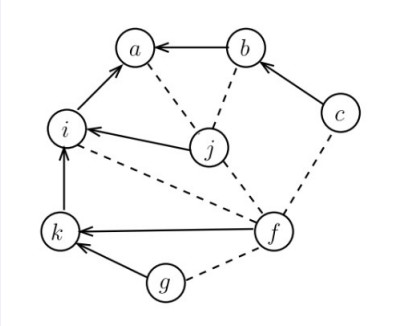
8: **when** BACK($data$) is received from $p_j$ such that $j \in children_i$ **do**
9: **begin:**
10: $val\_set_i = \bigcup_{j \in children_i} val\_set_j \cup \{(i, v_i)\}$
11: **if** $parent_i \neq p_i$ **then**
12: send BACK($val\_set_i$) to $parent_i$
13: **else**
14: the root $p_s$ can compute $f(val\_set_i)$
15: **end if**
16: **end**

**OBSERVACIÓN.** ¿Cómo cambiaría el algoritmo de convergecast en un sistema asíncrono?

**DEFINICIÓN 5: Árbol generador enraizado (Rooted Spanning Tree)**

Un árbol generador enraizado en $p_a$ es un árbol que contiene $n$ procesos y cuyos canales (aristas) son canales de la gráfica de comunicación. Cada proceso $p_i$ tiene un único padre, denominado localmente $parent_i$ , y un conjunto (posiblemente vacío) de hijos, denominado localmente $children_i$. Para simplificar la notación, el padre de la raíz es la propia raíz, es decir, el proceso distinguido $p_a$ es el único proceso $p_i$ tal que $parent_i = i$. Además, si $j \neq a$, tenemos $j \in children_i \iff parent_j = i$, y el canal $< i, j >$ pertenece a la gráfica de comunicación.

En la imagen se describe un ejemplo de árbol generador enraizado. Las flechas (orientadas hacia la raíz) describen los canales de la gráfica de comunicación que pertenecen al árbol generador. Las aristas punteadas son los canales de la gráfica de comunicación que no pertenecen al árbol generador. Este árbol generador enraizado en $p_a$ es tal que, al considerar la posición del proceso $p_i$ donde $neighbors_i = \{a, k, j, f\}$, tenemos $parent_i = a$, $children_i = \{j, k\}$ y, en consecuencia, $parent_j = parent_k = i$. Además, $children_i \cup \{parent_i\} \subseteq vecinos_i = \{a, k, j, f\}$.



**Algorithm 4** Algoritmo donde $p_i$ conoce la gráfica de comunicación - Código para el proceso $p_i$

1: $vecinos_i, proc\_conocidos_i = \{i\}$
2: $canales\_conocidos_i = \{< i, j > | j \in vecinos_i\}$
3: **begin:**
4: **for** each $j \in vecinos_i$ **do**
5: send POSITION($i, vecinos_i$) to $j$
6: **end**

7: **when** POSITION($k, vecinos$) is received from neighbor $p_j$
8: **begin:**
9: **if** $k \notin proc\_conocidos_i$ **then**
10: $proc\_conocidos_i = proc\_conocidos_i \cup \{k\}$
11: $canales\_conocidos_i = canales\_conocidos_i \cup \{< k, l > | l \in vecinos\}$
12: **for** $l \in vecinos_i - \{j\}$ **do**
13: send POSITION($k, vecinos$) to $l$
14: **end for**
15: **if** $\forall < l, m > \in canales\_conocidos_i : \{l, m\} \subset proc\_conocidos_i$ **then**
16: $p_i$ conoce la gráfica de comunicación **return**
17: **end if**
18: **end if**
19: **end**

**Algorithm 5** Algoritmo para construir un árbol generador - Código para el proceso $p_i$

```
1: Initially do
2: begin:
3:     if p_s = p_i then
4:         parent_i = i; expected_mssg_i = |neighbors_i|
5:         for each j ∈ neighbors_i do
6:             send GO() to p_j
7:         end for
8:     else
9:         parent_i = 0
10:    end if
11:    children_i = 0
12: end

13: when GO() is received from p_j do
14: begin:
15:    if parent_i = 0 then
16:        parent_i = j; expected_mssg_i = |neighbors_i| − 1
17:        if expected_mssg_i = 0 then
18:            send BACK(i) to p_j
19:        else
20:            for each k ∈ neighbors_i − {j} do
21:                send GO() to p_k
22:            end for
23:        end if
24:    else
25:        send BACK(0) to p_j
26:    end if
27: end

28: when BACK(val_set) is received from p_j do
29: begin:
30:    expected_mssg_i = expected_mssg_i − 1
31:    if val_set ≠ 0 then
32:        children_i = children_i ∪ {j}
33:    end if
34:    if expected_mssg_i = 0 then
35:        if parent_i ≠ i then
36:            send BACK(i) to parent_i
37:    end if
38: end
```

**Algorithm 6** Broadcast y Convergecast sobre un árbol generador - Código para el proceso $p_i$

1: Initially do
2: **begin:**
3: **if** $p_s = p_i$ **then**
4: $parent_i = i; expected\_mssg_i = |neighbors_i|$
5: **for** each $j \in neighbors_i$ **do**
6: send GO($data$) to $p_j$
7: **else**
8: $parent_i = 0$
9: **end if**
10: $children_i = 0$
11: **end**

12: **when** GO($data$) is received from $p_j$ **do**
13: **begin:**
14: **if** $parent_i = 0$ **then**
15: $parent_i = j; expected\_mssg_i = |neighbors_i| - 1$
16: **if** $expected\_mssg_i = 0$ **then**
17: send BACK($(i, v_i)$) to $p_j$
18: **else**
19: **for** each $k \in neighbors_i - \{j\}$ **do**
20: send GO($data$) to $p_k$
21: **end if**
22: **else**
23: send BACK(0) to $p_j$
24: **end if**
25: **end**

26: **when** BACK($val\_set$) is received from $p_j$ **do**
27: **begin:**
28: $expected\_mssg_i = expected\_mssg_i - 1$
29: **if** $val\_set \neq 0$ **then**
30: $children_i = children_i \cup \{j\}$
31: **end if**
32: **if** $expected\_mssg_i = 0$ **then**
33: $val\_set_i = \bigcup_{x \in children_i} val\_set_x \cup \{(i, v_i)\}$
34: **if** $parent_i \neq i$ **then**
35: send BACK($(i, v_i)$) to $parent_i$
36: **else**
37: the root $p_s$ can compute $f(val\_set_i)$
38: **end if**
39: **end**

**OBSERVACIÓN.** En sistemas síncronos, el algoritmo de inundación (flooding algorithm) es un método sencillo pero eficaz para construir un árbol generador BFS. Sin embargo, en sistemas asíncronos, el árbol generador construido por el algoritmo de inundación puede estar lejos de ser BFS.

**OBSERVACIÓN.** La búsqueda en profundidad (DFS) y el recorrido en profundidad (DFT) suelen referirse al mismo proceso algorítmico para explorar una gráfica o una estructura de árbol de una forma específica.
En ambos casos, el algoritmo comienza en un nodo concreto y luego visita recursivamente cada uno de sus hijos, profundizando en la estructura hasta llegar a un nodo hoja (es decir, un nodo sin hijos). A continuación, retrocede hasta el último nodo con hijos no explorados y continúa el proceso hasta haber visitado todos los nodos.
Así, aunque los términos DFS y DFT pueden utilizarse indistintamente, DFS tiende a utilizarse más en el contexto de la búsqueda de un nodo o camino específico en un grafo, mientras que DFT puede utilizarse de forma más general para describir el recorrido de un grafo sin ningún objetivo específico en mente.

**OBSERVACIÓN.** Algoritmo BFS

```
when START() is received do    % only the distinguished process receives this message %
(1)  send GO(−1) to itself.

when GO(d) is received from p_j do
(2)  if (parent_i = ⊥)
(3)     then parent_i ← j; children_i ← ∅; level_i ← d + 1;
(4)          expected_msg_i ← |neighbors_i \ {j}|;
(5)          if (expected_msg_i = 0)
(6)             then send BACK(yes, d + 1) to p_parent_i
(7)             else for each k ∈ neighbors_i \ {j} do send GO(d + 1) to p_k end for
(8)          end if
(9)     else if (level_i > d + 1)
(10)            then parent_i ← j; children_i ← ∅; level_i ← d + 1;
(11)                 expected_msg_i ← |neighbors_i \ {j}|;
(12)                 if (expected_msg_i = 0)
(13)                    then send BACK(yes, level_i) to p_parent_i
(14)                    else for each k ∈ neighbors_i \ {j} do send GO(d + 1) to p_k end for
(15)                 end if
(16)            else send BACK(no, d + 1) to p_j
(17)         end if
(18) end if.

when BACK(resp, d) is received from p_j do
(19) if (d = level_i + 1)
(20)    then if (resp = yes) then children_i ← children_i ∪ {j} end if;
(21)         expected_msg_i ← expected_msg_i − 1;
(22)         if (expected_msg_i = 0)
(23)            then if (parent_i ≠ i) then send BACK(yes, level_i) to p_parent_i
(24)                                   else p_i learns that the breadth-first tree is built
(25)                 end if
(26)         end if
(27) end if.
```

**Fig. 1.11** Construction of a breadth-first spanning tree without centralized control (code for $p_i$)

**OBSERVACIÓN.** Algoritmo DFT/DFS

```
when START() is received do    % only p_a receives this message %
(1)  parent_i ← i;
(2)  let k ∈ neighbors_i;
(3)  send GO({i}) to p_k; children_i ← {k}.

when GO(visited) is received from p_j do
(4)  parent_i ← j;
(5)  if (neighbors_i ⊆ visited)
(6)     then send BACK(visited ∪ {i}) to p_j; children_i ← ∅;
(7)     else let k ∈ neighbors_i \ visited;
(8)          send GO(visited ∪ {i}) to p_k; children_i ← {k}
(9)  end if.

when BACK(visited) is received from p_j do
(10) if (neighbors_i ⊆ visited)
(11)   then if (parent_i = i)
(12)          then the traversal is terminated        % global termination %
(13)          else send BACK(visited) to p_parent_i  % local termination %
(14)        end if
(15)   else let k ∈ neighbors_i \ visited;
(16)        send GO() to p_k; children_i ← children_i ∪ {k}
(17) end if.
```

**Fig. 1.17** Time and message optimal depth-first traversal (code for $p_i$)

**EJERCICIO 1.** Ejecuta el algoritmo BFS la figura 1.11 [libro de M. Raynal]) en la siguiente gráfica.

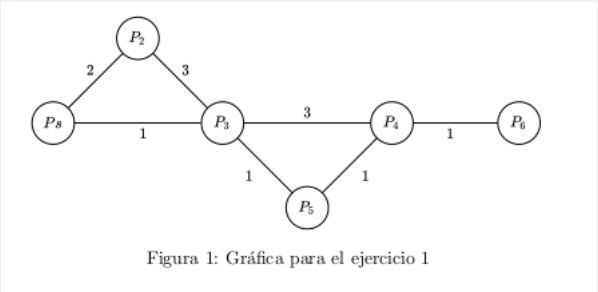
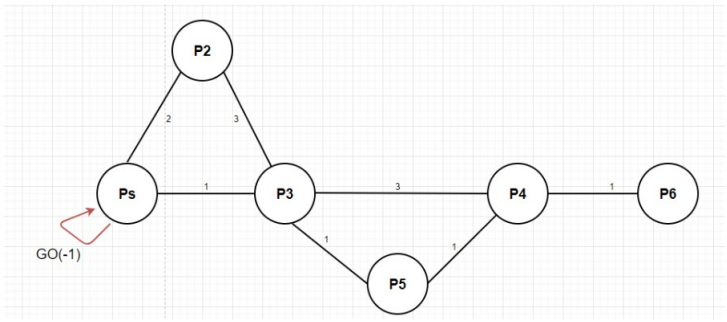


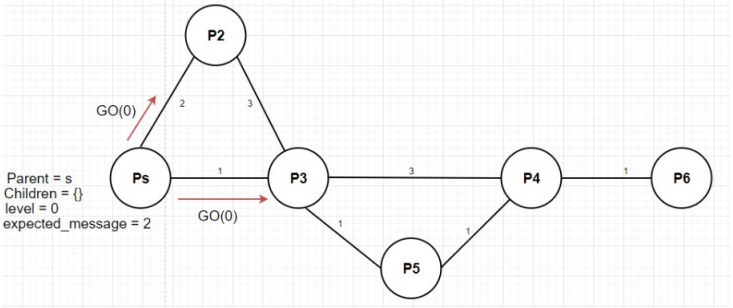

Figura 1: Gráfica para el ejercicio 1

*Solución.* Empezamos la ejecución en la ronda 0.

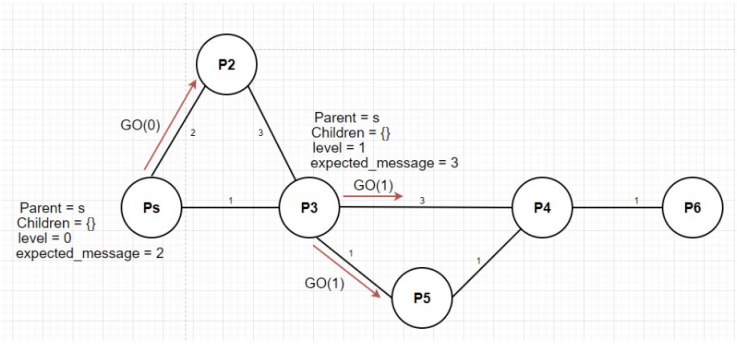
**Ronda 0:** PS al ser el proceso distinguido se envía a sí mismo el mensaje GO(-1).



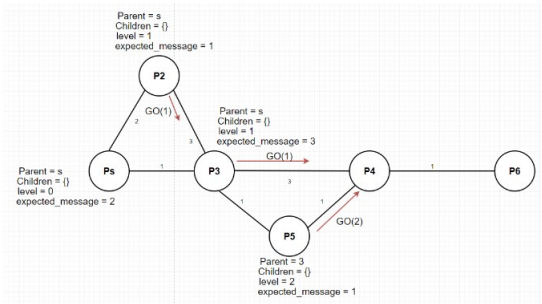
**Ronda 1:** PS manda los mensajes GO(0) a P3 y P2, sin embargo se tardará una ronda en llegar a P3 y 2 rondas en llegar a P2.



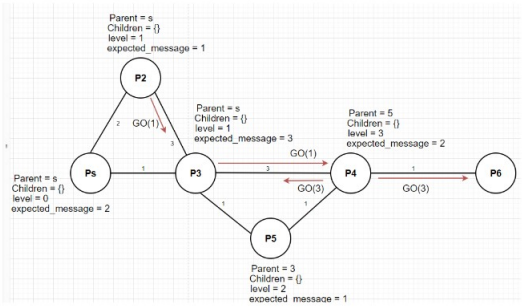

**Ronda 2:** Ya que el mensaje fue recibido por P3, este actualiza sus valores y manda un mensaje GO(1) a P4 y P5.



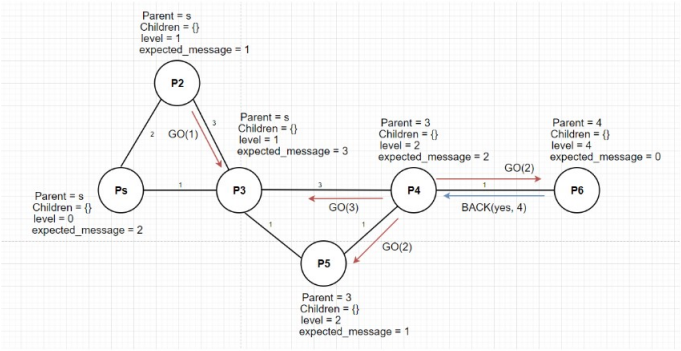

**Ronda 3:** El mensaje de Ps fue recibido por P2, así que P2 actualiza sus valores, también manda un mensaje GO(1) a P3. A su vez, P5 ya recibió el mensaje de P3 y por ende actualiza sus valores, finalmente P5 manda el mensaje GO(2) a P4.



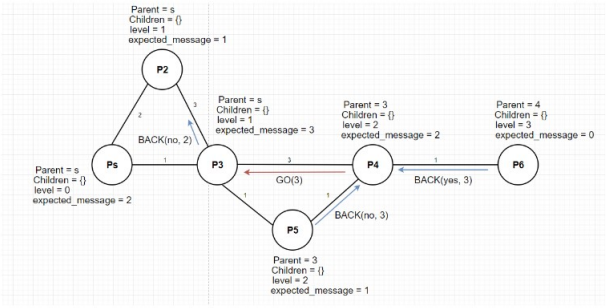

**Ronda 4:** El mensaje de P5 fue recibido por a P4, así que P4 actualiza sus valores, y manda un mensaje GO(3) a P6 y P3, el mensaje hacia P6 tardará una ronda y el mensaje a P3 tardará 3 rondas.



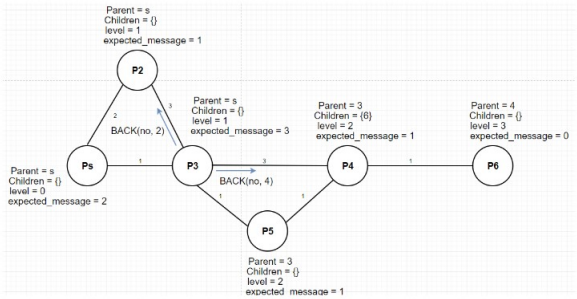
**Ronda 5:** El mensaje de P3 fue recibido por P4, P4 tenía de padre 5, sin embargo actualizara sus valores ya que su nivel será menor si su padre es P3, por lo que P4 envía a P6 el mensaje GO(2). Además, el mensaje anterior de P4 fue recibido por P6, así que P6 actualiza sus valores, pero como su $expected\_message = 0$, envía su mensaje BACK(yes, 4) a P4.



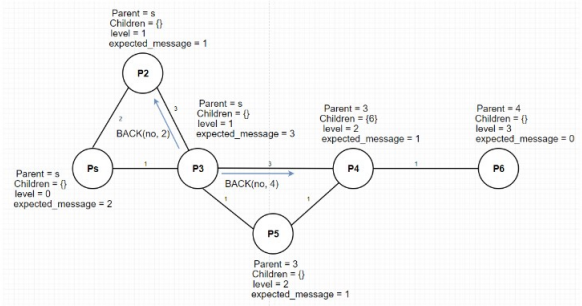

**Ronda 6:** El mensaje de P4 fue recibido por P5, pero al no cumplirse que el nivel sea mayor que d + 1, este manda un mensaje BACK(no,3) a P4, de igual forma, el mensaje de P2 fue recibido por P3 pero al no cumplirse que el nivel sea mayor que d+1, P3 manda un mensaje BACK(no, 2). Finalmente, el mensaje anterior de P4 lo recibe P6, y como el nivel es menor, este actualiza sus datos y envía el mensaje BACK(yes, 3) a P4.



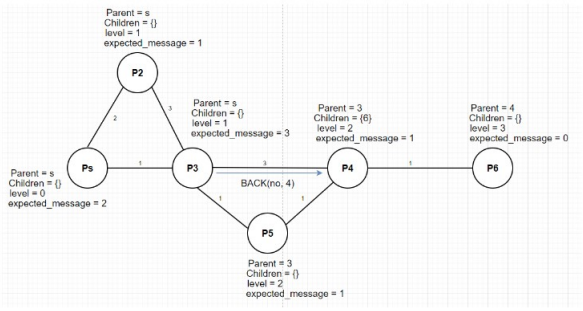

**Ronda 7:** El mensaje de P4 fue recibido por P3, pero al no cumplirse que el nivel sea mayor que d + 1, manda un mensaje BACK(no, 4). Además, P4 recibe el mensaje de P6, haciendo que su $expected\_message$ se reduzca por 1. P4 también recibe el mensaje de P5, pero al ser un BACK con un no, este lo ignora.



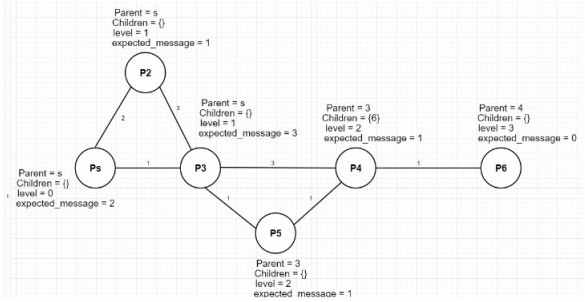

**Ronda 8:** Los mensajes de P3 hacia P2 y P4 aumentaron en una unidad de tiempo.



**Ronda 9:** El mensaje de P3 a P2 fue recibido, pero al ser un BACK con un no, P2 "ignora" el mensaje.




**Ronda 10:** El mensaje de P3 a P4 fue recibido, pero al ser un BACK con un no, P4 "ignora" el mensaje. Finalmente, el algoritmo no acaba debido a que el valor de *expected_message* de P4 no es cero, haciendo que nunca envíe un mensaje BACK con respuesta *yes* a sus vecinos P3 y P5.




□

**OBSERVACIÓN.** ¿Se puede obtener más de un árbol BFS en la gráfica anterior?

**OBSERVACIÓN.** Existen varios tipos de árboles generadores, cada uno con una finalidad distinta. Un árbol generador especialmente interesante es el árbol generador mínimo. El MST (minimum spanning tree) sólo tiene sentido en gráficas con pesos en las aristas, por lo que en esta sección supondremos que a cada arista $e$ se le asigna un peso $\omega e$.

**DEFINICIÓN 6: MST**

Dada un gráfica con pesos en las aristas $G = (V, E, \omega)$, el MST de $G$ es un árbol generador $T$ que minimiza $\omega(T)$, donde $\omega(G') = \sum_{e \in G'} \omega e$ para cualquier subgráfica $G' \subseteq G$.

**OBSERVACIÓN.** Supondremos que no hay dos aristas de la gráfica que tengan el mismo peso. Esto simplifica el problema, ya que hace que el MST sea único; sin embargo, esta simplificación no es esencial, ya que siempre se pueden romper los empates añadiendo los ID de los vértices adyacentes al peso.

**DEFINICIÓN 7: Aristas Azules**

Sea T un árbol generador de la gráfica con pesos $G$ y $T' \subseteq T$ una subgráfica de $T$ (también llamado fragmento). La arista $e = (u, v)$ es una arista saliente de $T'$ si $u \in T'$ y $v \notin T'$ (o viceversa). La arista saliente de peso mínimo $b(T')$ es la llamada arista azul de $T'$.

**LEMA 4.** Dada una gráfica con pesos en las aristas (ponderada) $G$ (tal que no hay dos pesos iguales), sea $T$ el MST, y $T'$ un fragmento de $T$. Entonces la arista azul de $T'$ también forma parte de $T$, es decir, $T' \cup b(T') \subseteq T$.

*Demostración.* Supongamos que en el MST $T$ hay una arista $e \neq b(T')$ que conecta $T'$ con el resto de $T$. Añadiendo la arista azul $b(T')$ al MST $T$ obtenemos un ciclo que incluye tanto a $e$ como a $b(T')$. Si eliminamos $e$ de este ciclo seguimos teniendo un árbol generador, y como por la definición de la arista azul $\omega_e > \omega_{b(T')}$, el peso de ese nuevo árbol generador es menor que el peso de $T$. Por lo tanto tenemos una contradicción. □

**OBSERVACIÓN.** En otras palabras, las aristas azules parecen ser la clave de un algoritmo distribuido para el problema MST. Puesto que cada nodo es en sí mismo un fragmento del MST, cada nodo tiene directamente una arista azul. Todo lo que tenemos que hacer es hacer crecer estos fragmentos. Esencialmente se trata de una versión distribuida del algoritmo secuencial de Kruskal.

**OBSERVACIÓN.** En un momento dado, los nodos de la gráfica se dividen en fragmentos (subárboles enraizados del MST). Cada fragmento tiene una raíz, el ID del fragmento es el ID de su raíz. Cada nodo conoce a su padre y a sus hijos en el fragmento. El algoritmo funciona por fases. Al principio de una fase, los nodos conocen los ID de los fragmentos de sus nodos vecinos.

- M. Raynal, Distributed Algorithms for Message-Passing Systems, Sec. 1.1.2 - 1.7
- H. Attiya, Distributed Computing, Sec. 2.2 - 2.5