Notas de la clase 4 – recorridos de grafos y árboles

Francisco Soulignac

30 de marzo de 2019

Aclaración: este es un punteo de la clase para la materia AED3. Se distribuye como ayuda memoria de lo visto en clase y, en cierto sentido, es un reemplazo de las diapositivas que se distribuyen en otros cuatrimestres. Sin embargo, no son material de estudio y no suplanta ni las clases ni los libros. Peor aún, puede contener "herrorez" y podría faltar algún tema o discusión importante que haya surgido en clase. Finalmente, estas notas fueron escritas en un corto período de tiempo. En resumen: **estas notas no son para estudiar sino para saber qué hay que estudiar**.

Tiempo total: 190 minutos

1. Recorrido de un grafo I: DFS (30 mins)

1.1. Componentes conexas

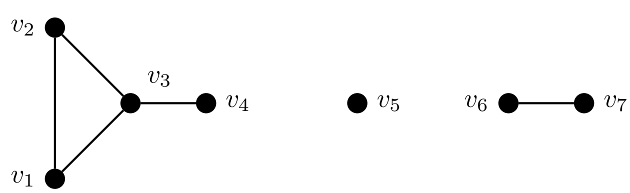
- Un grafo G es *conexo* cuando existe un camino entre todo par de vértices v y w. Gráficamente, podemos unir los puntos de v y w moviéndonos por líneas.

- Un conjunto de vértices V es *conexo* cuando $G[V]$ es conexo. Si V es maximal (bajo conexidad), entonces V es una *componente conexa*. A las componentes conexas les decimos simplemente *componentes*. Notar que G es conexo si y sólo si tiene una única componente.

- A veces, también se le llama *componente (conexa)* al grafo $G[V]$ donde V es una componente. Al igual que para el término conexo, se entiende por contexto.

Componentes de un grafo

Un grafo con tres componentes: $\{v_1, v_2, v_3, v_4\}$, $\{v_5\}$ y $\{v_6, v_7\}$.



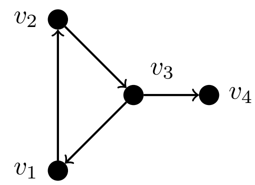
- Un digrafo es *conexo* cuando su grafo subyacente es conexo.

- Un digrafo es *fuertemente conexo* cuando existe un camino de v a w para todo par de vértices $v, w \in V(G)$.

- Un conjunto de vértices V es *fuertemente conexo* cuando $G[V]$ es fuertemente conexo. Si V es maximal (bajo conexidad fuerte), entonces V y $G[V]$ son *componentes fuertemente conexas*.

Componente fuertemente conexa

Un digrafo conexo con dos componentes fuertemente conexas: $\{v_1, v_2, v_3\}$ y $\{v_4\}$.



1.2. Recorrido DFS

- Digamos que un vértice w en un (di)grafo G es *alcanzable* por v cuando existe un camino de v a w.
- Dado un vértice v, se puede usar el siguiente algoritmo para "marcar" todos los vértices alcanzables por (o que alcanzan a) v.[1]
- Obviamente, además de marcar, se puede procesar cada vértice alcanzable.
- La lista `ordenDFS` mantiene el orden en que se marcan los vértices.

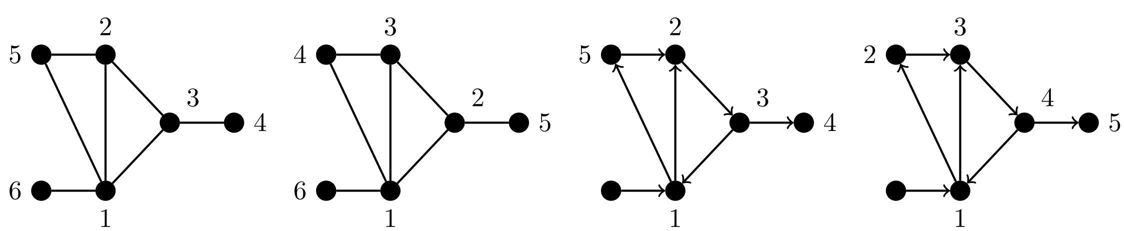
Algoritmo DFS recursivo para marcar los alcanzables

```
ordenDFS := secuencia vacia.
DFS(G, v, m):
  Si v está marcado, retornar ∅
  Marcar v con m y agregarlo v al final de ordenDFS
  Para cada w ∈ N_G^(±)(v): DFS(G, w)
```

- El algoritmo se llama *DFS* por sus siglas en inglés: "depth first search".
- Al orden en que se marcan los vértices se lo llama *orden DFS*.
- Vale notar que G tiene muchos ordenes DFS, que dependen del orden en que se revisa cada vecindario (más allá de si se revisa el vecindario total, de entrada o de salida).

Recorrido DFS

Distintos órdenes DFS desde un vértice 1; sólo los vértices etiquetados son marcados.



[1] En el caso en que G es un grafo, el algoritmo marca toda la componente de v. Cuando G es un digrafo marca más que la componente fuertemente conexa que contiene a v.

Teorema 1. *Sea v un vértice de un (di)grafo G y V el conjunto de vértices alcanzables por v. Si ningún vértice de V está marcado, entonces* DFS(G,v,m) *marca con m exactamente una vez a cada vértice de V. Si $N_G^{(+)}(w)$ se puede recorrer en tiempo $O(d(w))$ para todo w y la marca m se puede consultar y aplicar en $O(1)$ tiempo, entonces la complejidad temporal de* DFS(G,v,m) *es $O(\sum_{w \in V} d(v)) = O(n+m)$.*

Demostración. Claramente, la ejecución de DFS(G,v,m) no marca ningún vértice más de una vez, y la marca de todos los vértices marcados es m.

Por inducción en la cantidad de vértices marcados, podemos ver que si DFS(G,v,m) marca a w, entonces $w \in V$. En efecto, si todavía no se marcaron vértices es porque estamos en la primer llamada, i.e. $w = v$, y se marca $v \in V$. Para el paso inductivo, observemos que w se marca porque se realizó una llamada recursiva DFS(G,w,m) dentro de la aplicación de DFS(G,z,m) para algún $z \in V(G)$. Notar que $z \in V$ por hipótesis inductiva, i.e., existe un camino P de v a z. Más aún, como $w \in N^{(+)}(z)$, $P + w$ es un camino de v a w en G. Ergo, $w \in V$ como es requerido.

Veamos ahora que DFS(G,v,m) marca todos los vértices de V. Para ello, consideremos cualquier vértice $w \in V$ para el cual existe un camino $P = v_0, \ldots, v_k$ con $v = v_0$ y $w = v_k$. Veamos por inducción en k que w se marca. Esto es trivial cuando $k = 0$ porque $v = w$ se marca en la primer llamada. Para el paso inductivo, sabemos que v_{k-1} se marca en algún paso. En este momento se recorre $w = v_k \in N^{(+)}(v_{k-1})$ y se invoca a DFS(G,w,m). Con lo cual, si w no estaba marcado, se marca en esta llamada.

Finalmente, notemos que se recorre exactamente una vez $N^{(+)}(w)$ para cada w marcado. Por hipótesis, se requiere $O(d(w))$ tiempo para esta operación. En consecuencia, la complejidad de la instancia G, v, m es $t(G, v, m) = O(\sum_{w \in V} d(w)) = O(\sum_{w \in V(G)} d(w)) = O(n + m)$. □

- Para la implementación, lo único que hay que hacer es garantizar que $N_G^{(\pm)}(w)$ se pueda recorrer en $O(d(w))$ tiempo y la marca se puede aplicar y consultar en $O(1)$ tiempo.

- Resulta conveniente, pues, usar una lista de adyacencias para representar a G (que incluya $N^{(\pm)}$ cuando G es un digrafo) y un vector numérico M de n posiciones tal que $M(v)$ contenga la marca (un número) de v. Cada marca válida es un numero en $\{1, \ldots, n\}$, mientras que 0 se usa para desmarcar.

- Aplicado iterativamente, el algoritmo DFS permite marcar todas las componentes de un grafo G, usando una marca distinta para cada componente. Al algoritmo resultante también se lo llama DFS.

- El algoritmo DFS también se puede usar para encontrar las componentes fuertemente conexas de un digrafo. Queda para el laboratorio. Otras propiedades de DFS se estudian en PAP.

Algoritmo DFS para marcar todas las componentes

```
DFS(G):
  Desmarcar todos los vértices de G
  Para cada v ∈ V(G): DFS(G, v, v)
```

Teorema 2. *El algoritmo* DFS(G) *marca exactamente una vez todos los vértices de un grafo G, usando una marca distinta para cada componente. Su complejidad temporal es $O(n + m)$.*

Demostración. Sean $V_1, \ldots, V_k$ las componentes de G y v_i el primer vértices de V_i que se recorre. Por Teorema 1, v_i está desmarcado cuando se recorre y, por lo tanto, los vértices de V_i se marcan con la marca v_i luego de esta iteración. En consecuencia, la llamada DFS(G,w,w) no tiene efectos para $w \in V_i \setminus \{v_i\}$. En resumen, DFS($G$) marca exactamente una vez a cada $w \in V_i$ con la marca v_i.

En cuanto a la complejidad temporal, notemos que en $O(n+m)$ tiempo podemos representar a G usando lista de adyacencias (Observación de la clase anterior). Para las marcas, usamos un vector numérico M donde $M[v] = 0$ si v está desmarcado, mientras que $M[v] = x$ si v esta marcado con x. Desmarcar todos los vértices cuesta $O(n)$, mientras que determinar si un vértice está marcada y marcarlo cuesta $O(1)$. En este caso, la llamada `DFS(`G`,`w`,`w`)` cuesta $t(G, w, w) = O(1)$ tiempo para $w \in V_i \setminus \{v_i\}$. En consecuencia, por Teorema 1, el costo temporal es:

$$\begin{aligned}
\sum_{v \in V(G)} O(t(G, v, v)) &= O\left(\sum_{v \in V(G)} 1 + \sum_{i=1}^{k} t(G, v_i, v_i)\right) \\
&= O\left(n + \sum_{i=1}^{k} \sum_{v \in V_i} d(v)\right) \\
&= O\left(n + \sum_{v \in V(G)} d(v)\right) = O(n + m)
\end{aligned}$$

□

Corolario 1. *Se puede determinar si un grafo G es conexo en tiempo $O(n + m)$*

Demostración. Alcanza con chequear que todos los vértices tienen la misma marca luego de aplicar `DFS(`G`)`. □

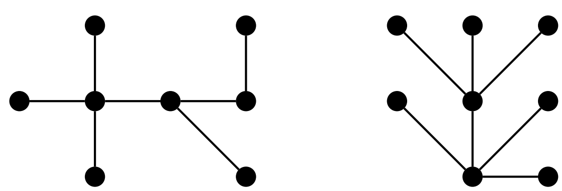
2. Árboles I (40 mins)

2.1. Definiciones básicas de árboles y bosques

- Un *bosque* es un grafo que no tiene ciclos.
- Si un bosque es conexo, entonces es un *árbol*.
- Una *hoja* es un vértice v con $d(v) = 1$.

Bosques y árboles

Un bosque con $n = 16$ vértices, $m = 14$ aristas y $k = 2$ árboles (componentes), cada uno de los cuales tiene $n = 8$ vértices, $m = 7$ aristas y $k = 1$ componente. Notar que en los tres casos $m = n - k$.



- Dado un grafo G cualquiera, el siguiente algoritmo encuentra una hoja o un ciclo.

Algoritmo para computar una hoja o un ciclo

```
leaf_or_cycle(G, v, w):
  Si d(w) = 1: retornar ([w], true)
  Si w está marcado: retornar ([w], false)
  Marcar w.
  Sea (C, b) = leaf_or_cycle(G, w, z) para z ∈ N(w) \ {v}
  Si b: retornar (C, b)
```

```
Si no: retornar (C + w, b') donde b' es verdadero sii el primero de C es w.
```

Lema 1. *Si G es un grafo sin vértices marcados, entonces `leaf_or_cycle(G,v,w)` retorna o bien una hoja $z \neq v$ de G o bien un ciclo de G para todo $vw \in E(G)$. Su complejidad temporal es $T(n) = O(n)$.*

Demostración. El algoritmo realiza una serie de llamadas recursivas que procesan los vértices $w_1, \ldots, w_k$ donde $w_1 = w$ y, si $k > 1$, $w_2 \neq v$. Por definición, w_i se marca cuando se procesa para cada $1 \leq i < k$. Luego, como el vértice w_{i+1} es un vecino no marcado de w_i, la secuencia $w_1, \ldots, w_k$ es un camino simple de G. Al procesar w_k, el algoritmo termina por uno de dos motivos. Si w_k está marcado, entonces $w_k = w_i$ para algún $i < k$. En este caso, el algoritmo retornar el par (w_k, false) que es recibido por la invocación anterior. Luego, para todo $i < j < k$, las sucesivas invocaciones retorna $(w_k, \ldots, v_j, \text{false})$. Finalmente, de la invocación i en adelante se retorna $(v_k, \ldots, v_i, \text{true})$, encontrándose efectivamente un ciclo. En cambio, si w_k no está marcado, el motivo de finalización es porque $d(w_k) = 1$, con lo cual w_k es una hoja. Si $k = 2$, entonces $w_2 \neq v$ por construcción; caso contrario, $w_k \neq v$ porque sino tendría dos vecinos w_1 y w_{k-1}. En cualquier caso, el algoritmo correctamente encuentra una hoja $w_k \neq v$. Finalmente, notemos que la complejidad es $O(n)$ porque se procesan a lo sumo n vértices y para cada vértices no hace falta revisar más que dos vecinos. ☐

Corolario 2. *Todo bosque con aristas tiene al menos dos hojas.*

Demostración. Por Lema 1, al aplicar `leaf_or_cycle(T,v,w)` para una arista arbitraria vw y un bosque T, se obtiene una hoja h de T. Más aún, por Lema 1, al aplicar `leaf_or_cycle(T,h,z)` para $z \in N(v_1)$ se obtiene una hoja $h' \neq h$ de T. ☐

Teorema 3. *Un grafo G es un bosque con k componentes si y sólo si:*

1. *G es un grafo sin aristas que tiene exactamente k vértices, o*
2. *G tiene una hoja v y $G - v$ es un bosque con k componentes.*

Demostración. Supongamos primero que G es un bosque con k componentes. Por Corolario 2, G tiene una hoja v. Claramente, $G - v$ es un bosque, ya que no se pueden generar ciclos cuando se elimina un vértice. Para ver que $G - v$ tiene k componentes, consideremos dos vértices u, w distintos a v que pertenecen a la misma componente. Por definición, existe un camino P de u a w en G. Claramente, todo vértice z de $P \setminus \{u, w\}$ tiene grado al menos 2 en G. Ergo, $z \neq v$ y, por lo tanto, P es un camino de $G - v$. En consecuencia, $G - v$ tiene k componentes.

Para la vuelta, observemos primero que cualquier grafo sin aristas es un bosque que, obviamente, tiene tantas componentes como vértices. Supongamos pues que G tiene una hoja v y que $G - v$ es un bosque con k componentes. Claramente, todo camino de $z \neq v$ a v en G contiene al único vecino w de v. Luego, z está en la misma componente que v en G si y sólo si z está en la misma componente que w en $G - v$. En consecuencia, G tiene la misma cantidad k de componentes que $G - v$. ☐

Corolario 3. *Si B es un bosque con k componentes, entonces $m = n - k$.*

Demostración. Porque si B no tiene aristas, entonces $m = 0$ y $n = k$, mientras que si B tiene una hoja v, entonces $B - v$ tiene $m - 1$ aristas, $n - 1$ vértices y k componentes. Luego, la demostración sale por inducción en la cantidad de aristas. ☐

- Problema de reconocimiento de bosques

  Input: un grafo G

  Output: si G es un bosque, un ordenamiento $v_1, \ldots, v_n$ tal que $d_H(v_i) \leq 1$ en $H = G[\{v_1, \ldots, v_i\}]$ para todo $1 \leq i \leq n$. Caso contrario, un ciclo de G.

- Notar que, por Corolario 3, G es un árbol si y sólo si $k = 1$.

Algoritmo para reconocer bosques

```
esBosque?(G):
  Crear una lista vacía S
  Mientras G tiene un vértice v con d(v) ≤ 1:
    Sacar v de G y agregarlo al inicio de S.
  Si G no tiene vértices: Retornar S
  Caso contrario: Retornar leaf_or_cycle(G, v, w) para vw ∈ E(G)
```

Teorema 4. *El algoritmo* `esBosque?` *resuelve el problema de reconocimiento de bosques y se puede implementar para que ejecute en tiempo $O(n)$.*

Demostración. El hecho de que `esBosque?` es correcto surge del Teorema 3 y del Lema 1; notar que `leaf_or_cycle(`G`,`v`,`w`)` encuentra un ciclo cuando G no tiene hojas.

Con respecto a la implementación, hacemos algunos precómputos a G antes de aplicar el algoritmo. El primer paso preliminar es eliminar aristas de G, dejando a lo sumo n de ellas. Para ello, alcanza con crear un nuevo grafo (usando lista de aristas) que contenga las n aristas. Este paso no cambia el resultado, dado que si G tiene $m \geq n$ aristas, entonces no es un bosque por Corolario 3. Suponemos pues, que G tiene $O(n)$ aristas desde ahora. Luego, G se transforma a una lista de adyacencias y se computa $d(v)$ para todo $v \in V(G)$. Estos pasos se pueden implementar en tiempo $O(n)$. Una vez hecho esto, se crea un conjunto (sobre vector) L que contenga las hojas de G. De esta forma, podemos tomar cualquier hoja v de L en tiempo $O(1)$. Durante el transcurso el algoritmo, $G = G_0$ se transforma eliminado iterativamente algunos vértices. Llamemos v_i al i-esimo vértice eliminado y $G_i = G_{i-1} - v_i$. Para computar $G_i = G_{i-1} - v_i$, no eliminamos físicamente ni a v_i ni a su arista v_iw de G_{i-1}. En su lugar, simplemente sacamos a v de L, decrementamos en 1 a $d(v_i)$ y a $d(w)$ y, en caso que $d(w) = 1$, insertamos w en L. Notar que v_i queda con grado $d(v_i) = 0$ aunque, como la eliminación es virtual, la lista de adyacencias de v_i contiene $d_G(v_i)$ vértices. Sin embargo, como $d(v_i) = 1$ antes de esta iteración, sabemos que w es el único que no está marcado con $d(w) > 0$. En consecuencia, el computo de G_i requiere $O(d(v_i))$ tiempo. Luego, el primer while requiere $O(n + m) = O(n)$ tiempo. Para saber si el grafo G_k obtenido al final tiene aristas, alcanza con revisar si $d(v) = 0$ para todo v. En caso negativo, aplicamos `leaf_or_cycle(`H`,`x`,`y`)` para una arista xy, teniendo cuidado de nunca visitar un vértice de grado 0. Como resultado obtenemos un ciclo de G en tiempo $O(n)$ por Lema 1. □

3. Intervalo (10 mins)

4. Árboles II (60 mins)

4.1. Reconocimiento con DFS

- Por Corolario 3, si 1. G es un bosque con 2. k componentes, entonces 3. $m = n - k$. La vuelta no vale, porque un grafo con $m = n - k$ aristas no tiene por qué ser un bosque. Pero, si combinamos dos de las tres condiciones siempre se obtiene un bosque.

Teorema 5. *Para un grafo G, son equivalentes:*

1. *G es un bosque con k componentes*
2. *G es un bosque con $m = n - k$ aristas*
3. *G tiene k componentes y $m = n - k$ aristas.*

Demostración. $1 \Rightarrow 2$. Como G tiene k componentes, $m = n - k$ por Corolario 3.

$2 \Rightarrow 3$. Es consecuencia directa del Teorema 3.

$3 \Rightarrow 1$. Por inducción en m. Si $m = 0$, entonces G es un bosque con $k = n$ componentes. Para el paso inductivo, notemos que como $\sum_{v \in V(G)} d(v) = 2m = 2(n-k)$, entonces G tiene al menos k vértice de grado menor a 2. Como G tiene k componentes, alguno de estos vértices, digamos v, tiene $d(v) > 0$. Es decir que v es un hoja. Por hipotesis inductiva, dado que $G - v$ tiene k componentes, $n-1$ vértices y $m-1 = (n-1)-k$ aristas, obtenemos que $G - v$ es un bosque con k componentes. Luego, G es un bosque con k componentes por Teorema 3. $\square$

- El Teorema 5 implica un segundo algoritmo para resolver el problema de reconocimiento de bosques.
- El algoritmo es correcto y requiere $O(n)$ tiempo.

Algoritmo DFS para reconocer bosques

```
esBosque?(G):
  Aplicar DFS(G) para contar la cantidad k de componentes.
  Retornar verdadero sii m = n - k.
```

- La ventaja de este algoritmo es que DFS tiene más aplicaciones por las que conviene implementarlo y, por lo tanto, suele estar implementado en bibliotecas.
- Recordemos que, en caso que G sea un bosque, hay que retornar un orden $v_1, \ldots, v_n$ tal que $d_H(v_i) \leq 1$ si $H = G[\{v_1, \ldots, v_i\}]$ para todo $1 \leq i \leq n$. Análogamente, hay que retornar un ciclo cuando G no es un bosque.
- Dejamos la respuesta a estos problemas para la Sección 4.3.
- Las siguientes caracterizaciones de los bosques resultan útiles más adelante y sirven para entender mejor su estructura.

Teorema 6. *Para un grafo G con k componentes son equivalentes:*

1. *G es un bosque,*
2. *existe a lo sumo un camino simple de v a w para todo $v, w \in V(G)$,*
3. *$G - vw$ tiene $k+1$ componentes para todo $vw \in E(G)$,*
4. *Si v y w están en la misma componente, entonces $G + vw$ tiene exactamente un ciclo que contiene a vw para todo $vw \notin E(G)$.*

Demostración. $\neg 2 \Rightarrow \neg 1$. Si G tiene dos caminos simples entre v y w, entonces G tiene un ciclo (ejercicio 5.8 de la guía).

$2 \Rightarrow 3$. Claramente, v, w es el único camino simple de v a w para todo $vw \in E(G)$. Ergo, v y w están en distintas componentes de $G - vw$.

$\neg 1 \Rightarrow \neg 3$. Si G tiene un ciclo $C = v_1, \ldots, v_k, v_1$, entonces para cualquier camino P que contiene a $v_1 v_k$ existe un camino P' que une los mismos vértices y que consiste en reemplazar $v_1 v_k$ por $v_1, \ldots, v_k$. Ergo, $G - vw$ tiene k componentes.

$2 \Rightarrow 4$. Como existe un camino simple $P = v_1, \ldots, v_k$ entre $v = v_1$ y $v_k = w$, entonces $G + vw$ tiene un ciclo $P + v_1$. Por otra parte, todos los ciclos de G contienen a vw; caso contrario el ciclo pertenece también a G lo que contradice $2 \Rightarrow 1$. Finalmente, si existieran dos ciclos $v_1, \ldots, v_k, v_1$ y $Q = w_1 \ldots, w_q, w_1$ en G con $v_1 = w_1 = v$ y $v_k = w_q = w$, entonces $v_1, \ldots, v_k$ y $w_1, \ldots, w_k$ son caminos simples que no contienen a vw. Pero esto contradice $2 \Rightarrow 1$ por ejercicio 5.8 de la guía.

$\neg 1 \Rightarrow \neg 4$. Si G tiene un ciclo C, entonces $G + vw$ tiene un ciclo que no contiene a vw para cualquier $vw \notin E(G)$. $\square$

4.2. Árboles y bosques enraizados

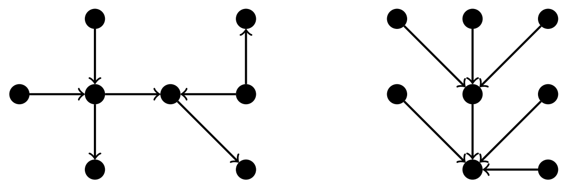
- Recordemos que un *grafo orientado* es un digrafo D donde $vw \in E(D)$ sólo si $wv \notin E(D)$.

- Una *orientación* de un grafo G es un grafo orientado D cuyo grafo subyacente es G. Gráficamente, D se obtiene de reemplazar las líneas de las aristas por flechas.

- Un *bosque orientado* no es más que un orientación T' de un bosque T. A priori, un bosque T admite una cantidad exponencial de orientaciones. En esta sección estamos interesados en los bosques y árboles enraizados.

Árboles orientados y enraizados

Dos árboles orientados. El de la derecha es, además, enraizado.



Observación 1. *Todo bosque orientado tiene al menos un vértice con grado de salida (resp. entrada) 0.*[2]

Demostración. Porque si v es una hoja con $d^{\text{out}}(v) = 1$, entonces $d^{\text{out}}_G(w) = d^{\text{out}}_H(w)$ para $H = G - v$. Luego, el resultado vale porque los bosques sin aristas tienen un vértice con $d^{\text{out}} = 0$. $\square$

- Un *árbol enraizado* T es un árbol orientado que contiene un único nodo r con $d^{\text{out}}(r) = 0$.

- El vértice r es la *raíz* de T. Por comodidad, en lugar de decir que T es un árbol enraizado con raíz r, decimos que T es un árbol con raíz r.

- Un *bosque enraizado* es un bosque orientado cuyas componentes son árboles enraizados. Sus *raíces* son los vértices con grado de salida 0.

- En la figura de arriba, el árbol orientado de la izquierda no es enraizado; el de la derecha sí.

Observación 2. *Si T es un árbol con raíz r, entonces $T - v$ es un árbol enraizado con raíz r para toda hoja $v \neq r$.*

Demostración. Porque $d^{\text{out}}_T(w) = d^{\text{out}}_R(w)$ para $R = T - v$. $\square$

Teorema 7. *Si T es un árbol con raíz r, entonces $d^{\text{out}}(v) = 1$ para todo $v \in V(T) \setminus \{r\}$. Más aún, el único vértice en $N^{\text{out}}(v)$ pertenece al único camino de v a r en T.*

Demostración. Consideremos un orden $v_1, \ldots, v_n$ de $V(T)$ tal que $v_1 = r$ y v_i es un hoja de $T(i) = T[\{v_1, \ldots, v_i\}]$ para todo $2 \leq i \leq n$. Notar que este orden existe por Teorema 3 y por el hecho de que todo árbol tiene al menos dos hojas (Corolario 2). Tomemos una arista cualquiera $v_i v_j \in E(T)$ y notemos que $j < i$. Caso contrario, entonces v_j es una hoja de $T(j)$ cuyo único vecino es v_i, contradiciendo el hecho de que v_1 es el único vértice con $d^{\text{out}} = 0$ en $T(j)$ (Observación 2). Pero entonces, como v_i tiene un único vecino den T, obtenemos que $N^{\text{out}}(v_i) = \{v_j\}$. Más aún, como el único camino de v_i a r en T es también un camino de $T(i)$, entonces v_j pertenece al único camino de v_i a v_j en el grafo subyacente de T. $\square$

Corolario 4. *Sea T un árbol enraizado. Entonces, r es la raíz de T si y sólo si existe un camino de v a r para todo $v \in V(T) \setminus \{r\}$.*

[2] Más adelante generalizamos este resultado.

- Por el teorema anterior, una vez elegida una raíz, existe una única forma de orientar un árbol para enraizarlo. Es por este motivo que la terminología para árboles (no dirigidos) aplica para árboles enraizados (e.g., hoja, grado, etc).

- Asimismo, a veces usamos árboles enraizados como si fueran árboles, en cuyo caso estamos pensando en el árbol subyacente (e.g., cuando decimos que un árbol enraizado T es un subgrafo de un grafo G).

- *Enraizar* es simplemente elegir una raíz, ya que la orientación es única.

- Una forma simple de representar un bosque enraizado T es codificando T con listas de adyacencias en donde se almacena sólo N^{out}.

- Pero, como $d^{\text{out}}(v) \leq 1$ para todo $v \in V(T)$, alcanza con guardar un diccionario P con n claves donde $P[v]$ indica el padre del vértice v.

- Obviamente, $P[r]$ no está definido cuando r es una raíz: se puede usar $P[r] = r$ o $P[r] = \perp$ para indicar esto.

- Esta representación es útil para aquellos algoritmos donde el output es un bosque, sea o no enraizado.

- Para operar con el bosque puede ser conveniente guardar también N^{in}, que se puede computar en $O(n)$ (ejercicio). Por ejemplo, teniendo en cuenta que todos los vértices alcanzan alguna raíz, se puede recorrer todos los vértices de G con DFS desde las raíces si se cuenta con N^{in}.

- En C++, la representación de un bosque T con raíces es la siguiente, donde usamos vector para representar el diccionario:

Representación simple de un árbol orientado

```
vector<int> parent(n);
//for(auto r: raíz) parent[r] = r;
```

- Dado que los árboles enraizados equivalen a las estructuras de datos comúnmente usadas, se puede definir toda la terminología conocida. Solo para ilustrar, van algunas definiciones sobre un árbol T con raíz r.

- El *padre* de un vértice es su único vecino de salida. Su *abuelo* es el padre del padre, etc. Sus *hijos* son los vértices de los que es padre. Los hijos de los hijos son los *nietos*, etc.

- Formalmente, el 0-ancestro de v es v y el k-ancestro de v es el padre del $(k-1)$ ancestro. Análogamente, los k-descendientes de v son los vértices tales que v es su k-ancestro.

- El conjunto de *ancestros* (*descendientes*) está formado por todos los k-ancestros (k-descendientes).

- El *nivel* de v es $\ell(v) = d_T(r, v)$.

- La *altura* de T es $h(T) = \max\{\ell(v) \mid v \in V(T)\}$.

- T es (resp. exactamente) *m-ario* cuando $d^{\text{out}}(v) \leq m$ (resp. $d^{\text{out}}(v) = m$) para todo v no hoja.

- T es *balanceado* cuando $\ell(v) \geq h(T) - 1$ para toda hoja v.

- T es *completo* cuando $\ell(v) = h(T)$ para toda hoja v.

- Un ejemplo de lema es el siguiente.

Lema 2. *Si T es un árbol m-ario con l hojas, entonces $l \leq m^{h(T)}$. Si T es completo y exactamente m-ario, entonces $l = m^{h(T)}$.*

4.3. Esquema general de recorrido: árboles DFS

- Recordemos que DFS(G, v, m) marca los vértices de G alcanzables por (o que alcanzan a) v.

- El siguiente es un esquema iterativo general para procesar estos vértices.

Esquema de recorrido de grafos

```
recorrer(G, v):
  Crear un vector P de n posiciones con P[v] = v. //P representa un árbol T
  Crear una secuencia V vacía y otra secuencia S = [(v, v)].
  Mientras S ≠ ∅:
    Eliminar el primer valor (w, z) de S.
    Si w ∉ V:
      Procesar w y agregarlo al final de V.
      Agregar (u, w) a S en algún orden, para todo u ∈ N^(±)(w).
```

- Con una demostración similar a la que hicimos para DFS, se puede demostrar que el esquema propuesto procesa una vez a cada vértice alcanzable. Más aún, V contiene los vértices procesados en el orden en que fueron procesados.

- Si se puede agregar $N^{(\pm)}(w)$ a S en $O(d(w))$ tiempo, entonces el algoritmo requiere $O(n+m)$ tiempo. En efecto, alcanza con guardar una marca para saber si $w \in V$ en $O(1)$ y el vecindario de w se recorre una única vez (cuando w se marca).

- Como invariante, además de que V contiene los vértices ya procesados, podemos notar que S contiene al menos un par (w, z) por cada $z \in V$ y cada $w \in N^{(\pm)}(z)$. Es decir que S contiene los vértices que ya se saben alcanzables pero que aún no fueron procesados.

- Notar que S puede contener un par (w, z) para algún w ya procesado. Más aún, S puede contener dos pares (w, z) y (w, z'). El orden en que se procesan depende del orden en que están en S.

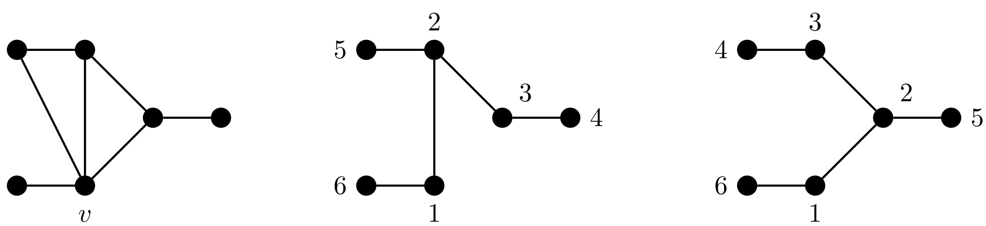
- El orden en que se agregan los elementos a S está indeterminado (porque es un esquema), al igual que el orden en que se revisan las aristas de $N^{(\pm)}(w)$.

- La secuencia P representa un árbol T con raíz v. Cada vez que se agrega un vértice w a V cuando se procesa el par (w, z), se agrega la arista wz a T. Esta arista indica que w fue “descubierto” primero por z.

- Cuando G es conexo y se usa $N(v)$, el árbol T es un *árbol generador* de G, ya que contiene a todos los vértices de G.

Árboles DFS

Un grafo y dos órdenes posibles con los respectivos árboles generadores que se obtienen invocando `recorrer(G,v)` cuando S se implementa usando una pila. Notar que los dos órdenes son DFS, lo que no es una casualidad, teniendo en cuenta que la recursión se implementa en código máquina con una pila.



- El algoritmo DFS es un caso particular de este esquema de recorrido en donde el conjunto S se implementa usando una pila.

- La pila S contiene las ramas activas de la recursión de acuerdo a su profundidad. Si $(w, z) \in S$, entonces en algún momento se invocó `recorrer(`G`,`z`)` y aún no se procesó la llamada recursiva `recorrer(`G`,`w`)`.

- En cada paso se elige el último vértices z que fue marcado (i.e., el último de la recursión) y se invoca recursivamente a DFS sobre todos sus vecinos. Como es la rama activa, los vecinos de z están en el tope de S. Tomándolos iterativamente se revisa $N^{(\pm)}(z)$.

- Más aún, el llamado recursivo con w es equivalente a marcar w y agregar su vecindario para procesar.

- Como corolario, DFS se puede computar iterativamente en $O(n+m)$ con el algoritmo anterior (personalmente, prefiero la recursión porque es más simple de implementar).

- A cualquier árbol T que se pueda obtener aplicando DFS(G, v) se lo llama *árbol DFS desde* v. Notar que se puede computar fácilmente con la implementación recursiva.

- Remarcamos que pueden existir muchos árboles DFS de G desde v (ver arriba), ya que el algoritmo no especifica en qué orden se debe recorrer cada $N^{(\pm)}(w)$ para agregarlo a S.

- El nombre DFS tiene que ver con el orden en que se procesan los vértices, en el que las ramas del árbol se construyen "primero a lo profundo".

- Finalmente, vale notar que si $v_1, \ldots, v_n$ es un orden DFS de G, entonces v_i es hoja en $T[\{v_1, \ldots, v_i\}]$. Ergo, si G es un árbol, el orden DFS vale como output de `esBosque?`.

- El siguiente lema permite encontrar un ciclo de G en el caso en que G no sea un bosque.

Lema 3. *Sea T un árbol DFS de un grafo G. Si $vw \in E(G) \setminus E(T)$ y el nivel de v es menor o igual al nivel de w, entonces v es un k-ancestro de w para $k \geq 2$. En consecuencia, $P + v$ es un ciclo de G para el único camino de v a w en T.*

Demostración. Porque w no se puede procesar antes de v. Caso contrario, como $vw \notin E(T)$, v es ignorado al recorrer $N(w)$. Esto sólo puede ocurrir si se procesa alguna llamada recursiva en la que w está activo. Pero en este caso v es descendiente de w, contradiciendo el hecho de que el nivel de v es menor al de w. Análogamente, cuando se procesa v y se considera vw, el mismo se descarta. Esto significa, dado que w se procesa luego de v, que w se procesa en un descendiente de v. En consecuencia, v es un k-ancestro de w para $k \geq 2$. ☐

- Se puede encontrar un ciclo en $O(n)$ tiempo cuando G no es un árbol.

- Con esto completamos nuestro algoritmo `esBosque?` que usa DFS. Queda como ejercicio.

5. Intervalo (10 mins)

6. Recorrido de un grafo II: BFS (20 mins)

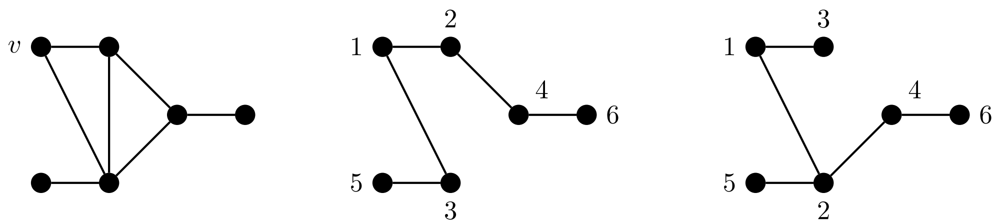
- Otra forma de recorrer un (di)grafo es procesando en cada paso el par (w, z) de S de forma tal de procesar antes los pares más viejos.

- Por el tipo de árbol que queda formado, a este algoritmo se lo llama BFS por sus siglas en inglés "breath first search".

- A cualquier árbol que se pueda obtener con BFS desde un nodo v se lo llama *árbol BFS (desde v)* de G.

Árboles y órdenes BFS

Un grafo con dos árboles BFS desde v junto a sus respectivos órdenes. Notar que la distancia desde v a cualquier otro vértice se preserva en los árboles.



- Para la implementación, alcanza con usar una cola para el conjunto S en lugar de una pila.

- Como encolar y desencolar toman $O(1)$ tiempo, BFS requiere $O(n+m)$ tiempo si G se implementa con listas de adyacencias.

Observación 3. *Sea T un árbol BFS desde r de un (di)grafo G. Si el nivel de v es menor al nivel de w en T, entonces v se procesó antes que w por BFS.*

Demostración. Por inducción en el nivel ℓ de v. El caso base $\ell = 0$ es trivial, porque r es el primer vértice que se procesa. Para el paso inductivo, sean p y q los padres de v y w en T, respectivamente. Por hipótesis inductiva, p se procesa antes que q. En consecuencia, el par (v, p) es generado antes que el par (w, q), por lo tanto, (v, p) se procesa antes por la regla de BFS. $\square$

Teorema 8. *Si T es un árbol BFS desde r de un (di)grafo G, entonces $d_G(r, v) = d_T(v, r)$ para todo v alcanzable desde r.*

Demostración. Por inducción en la distancia $d_G(v, r) = \ell$. El caso base $\ell = 0$ es trivial porque $d_G(r, r) = d_T(r, r) = 0$. Para el paso inductivo, consideremos un camino más corto $P = v_0, \ldots, v_\ell$ desde $v_0 = r$ a $v_\ell = v$ en G. Por hipótesis inductiva, $v_{\ell-1}$ está en el nivel $\ell - 1$. Cuando este vértice se procesa, se inserta el par $(v, v_{\ell-1})$ en el conjunto S. Si este par es considerado, entonces $vv_{\ell-1}$ es una arista de T y, por lo tanto, $d_T(v, r) = \ell$ como es requerido. En caso contrario, existe un par (v, w) que se procesa antes que $(v, v_{\ell-1})$. Notemos que este caso sólo es posible cuando $w \notin N(r)$ y, por lo tanto, $\ell > 1$ y w tiene un padre p. Por la regla BFS, (w, p) se procesó antes que $(v_{\ell-1}, v_{\ell-2})$. Por Observación 3, el nivel de w es menor o igual al nivel de v. Es decir que $d_T(r, w) < \ell$ y, por lo tanto, $d_T(v, r) \leq \ell$. Claramente, $d_T(v, r) \geq d_G(r, v)$ para cualquier árbol generador de G con raíz r. En consecuencia, $d_G(r, v) = d_T(v, r)$. $\square$

- Problema del camino más corto:

  Input: un (di)grafo G y un vértice v.

  Output: una estructura de datos T que acepta *queries de distancia* y *queries de camino*. Cada query de distancia toma como input un vértice w y retorna $d_G(v, w)$. Cada query de camino toma un vértice w y retorna un vértice z con $d_G(v, z) = d_G(v, w) - 1$. Ejecutando $d_G(v, w)$ queries de camino se computa el camino más corto.

- El teorema anterior nos dice que podemos resolver el siguiente problema de camino más corto en $O(n+m)$ tiempo para cualquier (di)grafo G. La estructura generada permite responder las queries de distancia y camino en $O(1)$ tiempo.

- El algoritmo consiste en computar el árbol BFS T de G desde v, representado con el vector de padres T.

- Por Teorema 8, $T[v]$ es una respuesta válida a la query de camino.

- Junto con cada vértice $w \in V(T)$ se computa de el nivel $d_T(w, v)$ de w que, por Teorema 8, coincide con $d_G(v, w)$.

- Ambas queries requieren $O(1)$ tiempo.

- Notar que muchas veces se requiere sólo el camino más corto entre un par de vértices v y w. En este caso, se puede cortar el algoritmo BFS cuando se encuentra w.

- Desafortunadamente, el algoritmo requiere $O(n + m)$ tiempo en peor caso. Si bien podría existir un algoritmo que tarde sólo $O(d(v, w))$ tiempo, nadie lo conoce.

7. Grafos bipartitos (20 mins)

- Un grafo es *bipartito* cuando $V(G)$ se puede particionar en dos conjuntos disjuntos V y W tal que $v \in V$ y $w \in W$ para todo $vw \in E(G)$.[3]

Grafos bipartitos

En ambos grafos se muestra una bipartición V, W donde V son los vértices pares y W los impares. El grafo de la derecha es $K_{2,4}$.

- Notar que la partición V, W es única si y sólo si G es conexo.

- Se suele denotar $G = (V, W, E)$ a un grafo bipartito con partición V, W (aunque podría tener otra partición de no ser conexo).

- El grafo $G = (V, W, E)$ es *bipartito completo* cuando $vw \in E$ para todo $v \in V$ y todo $w \in W$.

- Si $G[V \cup W]$ es bipartito completo para una partición V, W, entonces (V, W) es una *biclique*.

- Se suele denotar con $K_{i,j}$ a un grafo bipartito completo genérico que tiene particiones de tamaño i y j.

- Notar que $|E(K_{i,j})| = ij$: en consecuencia un grafo bipartito de n vértices puede tener hasta $n^2/4$ aristas.

Observación 4. *Para un grafo G, son equivalentes:*

1. *G es bipartito,*
2. *todo subgrafo de G es bipartito,*
3. *toda componente de G es bipartita.*

Demostración. $1 \Rightarrow 2$. Si $G = (V, W, E)$ es bipartito y $H = (V_H, E_H)$ es subgrafo de G, entonces $V \cap V_H$, $W \cap V_H$ es una bipartición de H porque toda arista de $E_H \subseteq E$ une un vértice de V con otro de W.

$2 \Rightarrow 3$. Es trivial porque las componentes de G son subgrafos de G.

$3 \Rightarrow 1$. Supongamos que G tiene componentes $G_1, \ldots, G_k$ con $G_i = (V_i, W_i, E_i)$ bipartito. Por definición, no hay aristas de V_i a W_j para todo $1 \leq i, j \leq j$. Luego, $V_1 \cup \ldots \cup V_k$ y $W_1 \cup \ldots \cup W_k$ forman una bipartición de G. $\square$

[3] Notar que una de las partes V o W podría ser vacía. Esto difiere de cómo se dio este concepto en semestres anteriores.

Teorema 9. *Para un grafo G, son equivalentes:*

1. *G es bipartito,*
2. *todos los ciclos de G tienen longitud par,*
3. *si T es un árbol generador enraizado de G, entonces toda arista de $E(G) \setminus E(T)$ une dos vértices cuyos niveles tienen distinta paridad.*

Demostración. $1 \Rightarrow 2$. Sea $G = (V, W, E)$ bipartito y consideremos un ciclo $C = v_1, \ldots, v_{k+1}$ con $v_1 = v_{k+1} \in V$. Por definición, $v_i \in V$ si y sólo si $v_{i+1} \in W$ para todo $1 \leq i \leq k$. Luego, por inducción, $v_{2i+1} \in V$ y $v_{2i} \in W$ para todo $1 \leq 2i \leq k$. En consecuencia, $v_k = v_{2i}$ para algún i, i.e., k es par.

$\neg 3 \Rightarrow \neg 2$. Supongamos que v y w están en niveles $2i + q$ y $2j + q$ para alguna arista $vw \in E(G) \setminus E(T)$ y algún $q \in \{0, 1\}$. Sea z el ancestro común de nivel máximo (quizá $z \in \{v, w\}$), que existe porque la raíz de T es un ancestro común, y supongamos que z está en el nivel p. Por definición, el único camino P_v de v a z tiene longitud $2i + q - p$, mientras que el único camino P_w de w a z tiene longitud $2j + q - p$. En consecuencia, el circuito C formado por P_v, P_w y vw tiene longitud impar $2(i + j + q - p) + 1$. Más aún, como z es el ancestro común de nivel mínimo, P_v y P_w no tienen vértices en común, i.e., C es un ciclo.

$3 \Rightarrow 1$. Alcanza con tomar la bipartición V, $W = V(G) \setminus V$, donde V son los vértices que están en niveles pares de T. □

- El teorema anterior implica un algoritmo de tiempo $O(n + m)$ para resolver de reconocer si un grafo es bipartito.

- Problema de reconocimiento de grafo bipartito

  Input: un grafo G

  Output: si G es bipartito, una bipartición V, W. Caso contrario, un ciclo de longitud impar.

- El algoritmo se describe a continuación; la implementación se discute sólo en el apéndice.

Algoritmo para reconocer si un grafo es bipartito

```
esBipartito?(G):
  Obtener un árbol generador T de G (usando DFS o BFS)
  Si los niveles de v y w tienen la misma paridad para vw ∈ E(G) \ E(T):
    Determinar el ancestro común z entre v y w de nivel más bajo
    Retornar P_v + P_w + v donde:
      - P_v es el camino de v a z en T
      - P_w es el camino de z a w en T
  Caso contrario:
    Retornar V, V(G) \ V donde V son los vértices en niveles pares de T.
```

- El concepto de grafo 2-partitos (i.e., bipartitos) se puede generalizar a grafos k-partitos. Un grafo es k-partito cuando sus vértices se pueden particionar en k conjuntos independientes.

- No se conocen algoritmos polinomiales para el problema de reconocer si un grafo es k-partito para ningún $k \geq 3$ fijo.

8. Comentarios adicionales

Valen los mismos comentarios bibliográficos que en la clase anterior.

Para esta clase el objetivo es aprender: 1. las formas usuales de recorrer un grafo, 2. qué son los árboles y los árboles enraizados y qué propiedades tienen, 3. cómo reconocer si un grafo es un árbol o un bosque y 4. conocer algunas propiedades de los árboles DFS y BFS. En el laboratorio y en Problemas, Algoritmos y Programación vemos más propiedades de DFS que permiten resolver otros problemas de grafos. En Problemas de Grafos y Tratabilidad Computacional se estudian variantes de BFS que permiten reconocer otras clases de grafos. Tanto en materias pasadas (AED2) como en futuras (e.g., Teoría de Lenguajes), se estudia cómo organizar la información en árboles para resolver distintos problemas.

A. Implementación de los algoritmos en C++

En esta sección implementamos algunos de los algoritmos en C++. El objetivo es mostrar cómo se traducen los algoritmos coloquiales a C++, eliminando posibles ambigüedades.[4]

A.1. Algoritmo DFS

Este es el algoritmo de la Sección 1.1. Supongamos que queremos escribir un programa que tome un grafo G por la consola. Vamos a suponer que G viene dado por listas de aristas. Es decir que el input comienza con dos valores n y m que indican $|V(G)|$ y $|E(G)|$, seguido por m líneas conteniendo cada una un par v, w que indica que $vw \in E(G)$. El primer paso del algoritmo es transformar la representación de G a una lista de adyacencias. Una vez hecho esto, invocamos al algoritmo DFS para marcar las componentes. Estas marcas se guardan en un vector numérico como se discute en la Sección 1.1. El resto de la implementación es la traducción del algoritmo DFS a C++.

Algoritmo DFS recursivo (componentes.cpp)

```
using graph = vector<vector<int>>; //representacion de listas de adyacencias
vector<int> component;             //component[i] es la marca del i-esimo vertice.
const int none = -1;               //usamos -1 para desmarcar

//algoritmo DFS(G, v, m)
void mark_component(const graph& G, int v, int m) {
  if(component[v] == none) {
    component[v] = m;
    for(auto w: G[v]) mark_component(G, w, m);
  }
}

//algoritmo DFS(G)
void mark_components(const graph& G) {
  component.assign(G.size(), none);
  for(int v = 0; v < G.size(); ++v) mark_component(G, v, v);
}

int main() {
  int n, m; cin >> n >> m;
  graph G(n);
  for(int e = 0; e < m; ++e) {
    int v, w; cin >> v >> w;
```

[4] Por falta de tiempo, los algoritmos fueron testeados superficialmente.

```
    G[v].push_back(w);
    G[w].push_back(v);
  }
  mark_components(G);
  for(int v = 0; v < n; ++v)
    cout << "component[" << v << "] = " << component[v] << endl;
  cout << endl;
  return 0;
}
```

A.2. Reconocimiento de bosques: versión I

Este es el algoritmo de la Sección 2.1. El input es una lista de aristas como en el ejemplo anterior. El algoritmo se divide en dos procedimientos principales. Por un lado, `eliminarHojas` elimina iterativamente las hojas en forma virtual como se explica en la Seccion 2.1. Retorna las hojas eliminadas en el orden esperado, junto con el vector de grados de los vértices resultantes. Por otro lado, `imprimirCiclo` se usa para imprimir el ciclo. Dado un vértice w con grado mayor a 0, busca un camino como en `leaf_or_cycle` (Sección 2.1), teniendo en cuenta de no visitar los vértices eliminados ni el vértice anterior v.

Reconocimiento de bosques sin DFS (esBosqueI.cpp)

```
using graph = vector<vector<int>>;

pair<vector<int>,vector<int>> eliminarHojas(const graph& G) {
  vector<int> d(G.size()), L, res;
  for(int v = 0; v < G.size(); ++v) {
    d[v] = G[v].size();
    if(d[v] == 0) res.push_back(v);
    if(d[v] == 1) L.push_back(v);
  }
  while(not L.empty()) {
    auto v = L.back(); L.pop_back();
    if(d[v] == 0) continue;
    d[v] -= 1; res.push_back(v);
    for(auto w: G[v]) if(d[w] > 0) {
      d[w] -= 1;
      if(d[w] == 0) res.push_back(w);
      if(d[w] == 1) L.push_back(w);
    }
  }
  return make_pair(res, d);
}

int imprimir_ciclo(const graph& G, const vector<int>& d, vector<bool>& visited, int v, int
  w) {
  if(visited[w]) {
      cout << w << " ";
      return w;
  }
  visited[w] = true;
  int z = 0; while(d[G[w][z]] == 0 or G[w][z] == v) z += 1;
  auto last = imprimir_ciclo(G, d, visited, w, G[w][z]);
  if(last >= 0) cout << w << " ";
```

```
  return w == last ? -1 : last;
}

int main() {
  //leer grafo G

  vector<int> ordenHojas, d;
  tie(ordenHojas, d) = eliminarHojas(G);

  //Imprimir output
  if(ordenHojas.size() == G.size()) {
    cout << "bosque: ";
    for(auto v: ordenHojas) cout << v << " "; cout << endl;
  } else {
    cout << "no bosque: ";
    vector<bool> visited(G.size(), false);
    int start = 0; while(d[start] == 0) start += 1;
    imprimir_ciclo(G, d, visited, -1, start);
    cout << endl;
  }
}
```

A.3. Reconocimiento de bosques: versión II

Este es el algoritmo que aplica DFS para determinar si un grafo G es un bosque (Sección 4.1). La parte central está en la función `DFS` que toma un vértice v, el vértice p desde el que se invocó a v, el vector de niveles calculados l, el árbol output T implementado con el vector de padres, y el vector B de las aristas que están en G y no en T. Un vértice esta marcado cuando su nivel es mayor a `none`. Observemos que si tenemos una arista $vp \in E(G) \setminus E(T)$, ésta arista se recorre dos veces, cuando se procesa v y cuando se procesa p. Supongamos que el nivel de p es menor que el de v. En este caso, cuando se invoque `DFS(G, v, p, ...)`, el vértice p, al no ser el padre de v, tiene un nivel menor a $l[v] - 1$. En este caso es que registramos la arista. Finalmente, en la función `main` en lugar de retornar el orden DFS, retornamos el árbol por simplicidad.

Reconocimiento de bosques con DFS (esBosqueII.cpp)

```
using graph = vector<vector<int>>; //representacion de listas de adyacencias
using edge = pair<int,int>;
using tree = vector<int>;
const int none = -1;   //usamos -1 para desmarcar

void DFS(const graph& G, int v, int p, vector<int>& l, tree& T, vector<edge>& B) {
  if(l[p] + 1 < l[v]) B.push_back({p,v});
  if(l[v] == none) {
    T[v] = p; l[v] = l[p]+1;
    for(auto w: G[v]) DFS(G, w, v, l, T, B);
  }
}

//algoritmo DFS(G)
pair<tree, vector<edge>> bosqueDFS(const graph& G) {
  vector<int> l(G.size(), none);
  tree T(G.size(), none);
```

```
  vector<edge> B;
  for(int v = 0; v < G.size(); ++v) DFS(G, v, v, 1, T, B);
  return make_pair(T, B);
}

int main() {
  //leer grafo G...

  tree T; vector<edge> B;
  tie(T, B) = bosqueDFS(G);

  if(B.empty()) {
    cout << "es bosque: ";
    for(int v = 0; v < n; ++v) cout << T[v] << " <- " << v << ", "; cout << endl;
  } else {
    cout << "no es bosque: " <<  B.front().first;
    for(auto v = B.front().second; v != B.front().first; v = T[v]) cout << " " << v;
    cout << " " << B.front().first << endl;
  }
}
```

A.4. Camino más corto

El algoritmo que sigue permite resolver el problema del camino más corto. El input es un grafo como antes, seguido del vértice v (que llamamos r) y un número q de queries. Luego, siguen q valores representado cada query, para lo que se retorna la distancia y el primer vértice del camino. El algoritmo BFS es una implementación directa del algoritmo discutido en la Sección 6, con la excepción de que, además del árbol T, se calcula también el nivel de cada vértice en d.

Algoritmo BFS para calcular el camino más corto (bfs.cpp)

```
using graph = vector<vector<int>>;
using tree = vector<int>;
const int none = -1;

pair<tree, vector<int>> BFS(const graph& G, int r) {
  tree T(G.size(), none); vector<int> d(G.size(),-1);
  deque<pair<int,int>> cola{{r,r}};
  while(not cola.empty()) {
    int v, p; tie(v,p) = cola.front();
    cola.pop_front();
    if(T[v] == none) {
      T[v] = p; d[v] = d[p] + 1;
      for(auto w: G[v]) cola.push_back({w,v});
    }
  }
  return make_pair(T, d);
}

int main() {
  //leer grafo G
  int q, r; cin >> q >> r;

```

```
  tree T; vector<int> d;
  tie(T, d) = BFS(G, r);

  for(int i = 0; i < q; ++i) {
    int v; cin >> v;
    cout << "d(" << v << ", " << r << "): " << d[v] << ". T[v]: " << T[v] << endl;
  }
}
```

A.5. Reconocimiento de grafos bipartitos

El algoritmo para reconocer grafos bipartitos se divide en dos partes: encontrar un árbol generador y verificar que todas las aristas van a niveles de distinta paridad. En la implementación, `bipartition` se encarga de calcular el árbol DFS de G mientras marca la paridad del nivel con 0 (nivel par) o 1 (nivel impar). En caso que una arista vw incida en vértices de partes distintas, `print_cycle` se encarga de imprimir el ciclo impar buscando el ancestro común de vw e imprimiendo ambos caminos (uno en reversa). Caso contrario, la bipartición se obtiene de acuerdo a la paridad de los niveles.

Reconocimiento de grafos bipartitos (esBipartito.cpp)

```
vector<int> part;       //part[i] marca al i-esimo vértice con su partición candidata.
vector<int> parent;     //arbol dfs

const int none = -1;    //usamos -1 para desmarcar
const int in_cycle = 2; //marca de pertenencia a ciclo impar
const int lca = 3;      //marca del ancestro comun mas bajo

//algoritmo DFS donde v se marca con un valor {0,1} distinto al de su padre
void bipartition(const graph& G, int v, int p = none) {
  if(part[v] == none) {
    part[v] = p == none ? 0 : 1-part[p];
    parent[v] = p;
    for(auto w: G[v]) bipartition(G, w, v);
  }
}

//algoritmo DFS(G)
void bipartition(const graph& G) {
  part.assign(G.size(), none);
  parent.assign(G.size(), none);
  for(int v = 0; v < G.size(); ++v) bipartition(G, v);
}

//Imprime un ciclo impar para una arista vw con part[v] != part[w]
void print_cycle(int v, int w) {
  for(int z = v; z != none; z = parent[z]) part[z] = in_cycle;
  vector<int> cycle;
  for(; part[w] != in_cycle; w = parent[w]) cycle.push_back(w);
  part[w] = lca;
  cout << w << ", ";
  for(auto x = cycle.rbegin(); x != cycle.rend(); ++x) cout << *x << ", ";
  for(; part[v] != lca; v = parent[v]) cout << v << ", ";
```

```
  cout << v << endl;
}

int main() {
  //leer grafo
  bipartition(G);

  for(int v = 0; v < n; ++v) for(auto w: G[v]) if(part[v] == part[w]) {
      cout << "Ciclo impar encontrado: "; print_cycle(v,w);
      return 0;
  }

  cout << "Biparticion: ";
  for(int b = 0; b < 2; ++b) {
    cout << "{"; for(int v = 0; v < n; ++v) if(part[v] == b) cout << v << " "; cout << "}";
  }
  cout << endl;
  return 0;
}
```