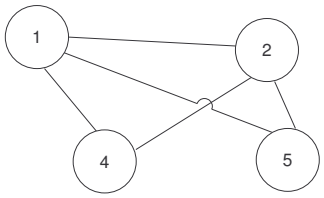
GRAFOS

GRAFOS

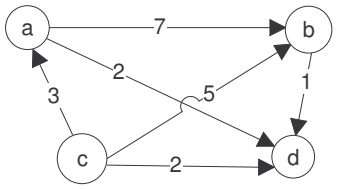
1. DEFINICIONES

El lector o lectora ya conocen el tipo de datos grafo porque ha sido introducido en otras asignaturas que preceden a ésta. Esta sección se va a dedicar a recordar la terminología que se emplea con ellos. En general, un grafo se utiliza para representar relaciones arbitrarias entre objetos del mismo tipo. Los objetos reciben el nombre de nodos o vértices y las relaciones entre ellos se denominan aristas. El grafo G, formado por el conjunto de vértices V y por el conjunto de aristas E, se denota por el par G=<V,E>.

Habitualmente distinguimos entre grafos dirigidos y no dirigidos, dependiendo de si las aristas están orientadas o no lo están, y entre grafos etiquetados o no etiquetados en función de si las aristas tienen o no información asociada. Gráficamente (los círculos representan los vértices y las líneas que los unen representan las aristas):



Grafo no dirigido y no etiquetado



Grafo dirigido y etiquetado

El tipo grafo que vamos a usar no permite aristas 'lazo' (una arista en que el mismo vértice es origen y destino), ni tampoco 'multiaristas' (dados dos vértices existe entre ellos más de una arista en el mismo sentido). Las operaciones que vamos a manejar van a ser las habituales de los conjuntos (crear, insertar elemento, eliminar elemento) sabiendo que tenemos elementos de dos tipos: vértices y aristas, y

operaciones específicas como grado, sucesores, adyacentes, etc. No seremos excesivamente rigurosos y haremos todas las salvedades que nos convengan para no tener que entrar en grandes detalles de implementación.

1.1. **Adyacencias**

Sea G=<V,E> un grafo NO DIRIGIDO. Sea v un vértice de G, $v \in V$. Se define:
- adyacentes de v, $\mathbf{ady}(v) = \{ v' \in V \mid (v,v') \in E \}$
- grado de v, $\mathbf{grado}(v) = | \mathbf{ady}(v) |$. Si un vértice está aislado, su grado es cero.

Sea G=<V,E> un grafo DIRIGIDO. Sea v un vértice de G, $v \in V$. Se define:
- sucesores de v, $\mathbf{suc}(v) = \{ v' \in V \mid (v,v') \in E \}$
- predecesores de v, $\mathbf{pred}(v) = \{ v' \in V \mid (v',v) \in E \}$
- adyacentes de v, $\mathbf{ady}(v) = \mathbf{suc}(v) \cup \mathbf{pred}(v)$
- grado de v, $\mathbf{grado}(v) = | |\mathbf{suc}(v)| - |\mathbf{pred}(v)| |$
- grado de entrada de v , $\mathbf{grado\_e}(v) = |\mathbf{pred}(v)|$
- grado de salida de v, $\mathbf{grado\_s}(v) = |\mathbf{suc}(v)|$

1.2. **Caminos**

Un CAMINO de longitud $n \geq 0$ en un grafo G=<V,E> es una sucesión $\{v_0, v_1, ..., v_n\}$ tal que :
- todos los elementos de la sucesión son vértices, es decir, $\forall i: 0 \leq i \leq n : v_i \in V$, y
- existe arista entre todo par de vértices consecutivos en la sucesión, o sea, $\forall i: 0 \leq i < n : (v_i, v_{i+1}) \in E$.

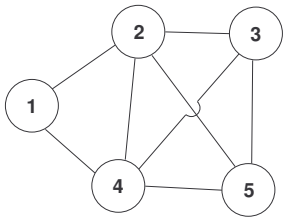
Dado un camino $\{v_0, v_1, ..., v_n\}$ se dice que :
- su LONGITUD viene dada por el número de aristas que lo forman y sus extremos son v_0 y v_n
- es PROPIO si $n > 0$. Equivale a que, como mínimo, hay dos vértices en la secuencia y, por tanto, su longitud es ≥ 1.
- es ABIERTO si $v_0 \neq v_n$
- es CERRADO si $v_0 = v_n$
- es SIMPLE si no se repiten aristas
- es ELEMENTAL si no se repiten vértices, excepto quizás los extremos. Todo camino elemental es simple.

Un CICLO ELEMENTAL es un camino cerrado, propio y elemental, es decir, es una secuencia de vértices, de longitud mayor que 0, en la que coinciden los extremos y no se repiten ni aristas ni vértices.

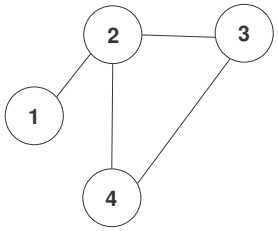
1. 3. **Conectividad**

Sea G=<V,E> un grafo NO DIRIGIDO. Se dice que:
- es CONEXO si existe camino entre todo par de vértices.
- es un BOSQUE si no contiene ciclos
- es un ARBOL NO DIRIGIDO si es un bosque conexo

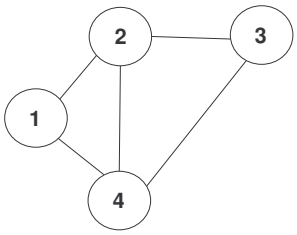
Un SUBGRAFO H=<U,F> del grafo G, es el grafo H tal que U⊆V y F⊆E y F⊆UxU.
Un ARBOL LIBRE del grafo G es un subgrafo, H=<U,F>, tal que es un árbol no dirigido y contiene todos los vértices de G, es decir, U=V. Los árboles libres son árboles (bosque conexo) sin un elemento distinguido o raíz y cualquier vértice del árbol puede actuar como tal.
Un SUBGRAFO INDUCIDO del grafo G es el grafo H=<U,F> tal que U⊆V y F contiene todas aquellas aristas de E tal que sus vértices pertenecen a U.
Ejemplo:



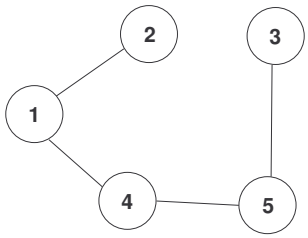
G=<V,E>



H=<U,F> subgrafo de G



H=<U,F> subgrafo inducido de G



H=<V,F> árbol libre de G

Sea G=<V,E> un grafo NO DIRIGIDO. Una COMPONENTE CONEXA de G es un subgrafo conexo inducido de G, H=<U,F>, tal que ningún otro subgrafo conexo inducido de G contiene a H. Dicho de otra forma, una componente conexa es un subgrafo conexo inducido MAXIMAL (es el más grande que se puede construir). Un grafo NO CONEXO G se puede partir de una sola forma en un conjunto de subgrafos conexos y cada uno de ellos es una COMPONENTE CONEXA de G.

Sea G=<V,E> un grafo NO DIRIGIDO, se dice que G es BIPARTIDO si todos sus vértices se pueden dividir en dos conjuntos disjuntos tal que todas las aristas enlazan 2 vértices en que cada uno de ellos pertenece a un conjunto distinto.

Sea G=<V,E> un grafo DIRIGIDO. Se dice que es FUERTEMENTE CONEXO si existe camino entre todo par de vértices en ambos sentidos.
Una COMPONENTE FUERTEMENTE CONEXA de G es un subgrafo fuertemente conexo inducido de G, H=<U,F>, tal que ningún otro subgrafo fuertemente conexo inducido de G contiene a H. Equivale a que una componente fuertemente conexa es un subgrafo fuertemente conexo MAXIMAL.
Se dice que G es DÉBILMENTE CONEXO si al convertir el grafo dirigido en uno no dirigido el grafo resultante es CONEXO. G es UNILATERALMENTE CONEXO si entre todo par de vértices *u* y *v*, hay un camino que une *u* con *v* o *v* con *u*.

1.4. **Algunos grafos particulares**

COMPLETO

Un grafo no dirigido G=<V,E> es COMPLETO si existe arista entre todo par de vértices de V. Sea n=|V|, el número de aristas de un grafo completo no dirigido es exactamente $|E|=n\cdot(n-1)/2$. Si se trata de un grafo dirigido entonces $|E|=n\cdot(n-1)$.

GRAFOS EULERIANOS

Se dice que un grafo no dirigido G=<V,E> es EULERIANO si existe un camino cerrado, propio, simple (no se repiten aristas) pero no necesariamente elemental que incluye todas las aristas de G.
Los siguientes lemas permiten determinar si un grafo dado es eulearíano:
Lema 1: Un grafo no dirigido y conexo es euleriano si y sólo si el grado de todo vértice es par.
Lema 2: Un grafo dirigido y fuertemente conexo es euleriano si y sólo si el grado de todo vértice es cero.
Averiguar si un grafo no dirigido y conexo es euleriano tiene un coste de $\theta(n)$, si se supone conocido el grado de cada vértice; gracias al lema 1 basta con recorrer el conjunto de vértices y comprobar si el grado de cada uno de ellos es par o no lo es.

GRAFOS HAMILTONIANOS

Un grafo no dirigido G=<V,E> es HAMILTONIANO si existe un camino cerrado y elemental (no se repiten vértices) que contiene todos los vértices de G. Si existe, el camino se llama circuito hamiltoniano.

En este caso no existe ninguna propiedad que permita determinar la existencia del camino por lo que averiguar si existe tiene el mismo coste que calcular directamente el camino (es un problema NP-C). La forma habitual de resolverlo es aplicar el esquema de *Vuelta Atrás* que va construyendo todos los caminos posibles y, para cada uno de ellos, comprueba si cumple la condición de hamiltoniano.

2. IMPLEMENTACIONES

Las implementaciones típicas de los grafos son 2: usando matrices de adyacencia y usando listas de adyacencia (o una variante con multilistas de adyacencia). A continuación daremos un breve repaso al coste de algunas de las operaciones habituales para cada una de estas implementaciones. Para facilitar la implementación vamos a suponer que los vértices se identifican con un número natural entre 1 y n, siendo n el número de vértices del grafo. De no ser así habrá que asociar a cada una de las etiquetas de los vértices un número natural que funcionará como una clave. Una tabla de hash puede ser útil para hacer esa asociación.

2.1. **Matrices de adyacencia**

Sea G=<V,E> y sea n=|V|. Supongamos que tenemos el grafo G implementado en una matriz de booleanos M[1..n,1..n] de modo que ($\forall v,w \in$ V: M[v,w]=CIERTO $\Leftrightarrow$ (v,w)$\in$ E). Si el grafo es etiquetado en vez de una matriz de booleanos usaremos una matriz del tipo de las etiquetas del grafo.

El espacio ocupado por la matriz es del orden de $\theta(n^2)$. Un grafo no dirigido sólo necesita la mitad del espacio, $(n^2-n)/2$, y uno dirigido lo necesita todo excepto la diagonal, es decir, n^2-n.

El coste temporal de las operaciones básicas del grafo varía entre $\theta(1)$ y $\theta(n^2)$. En un grafo no dirigido los costes serán:

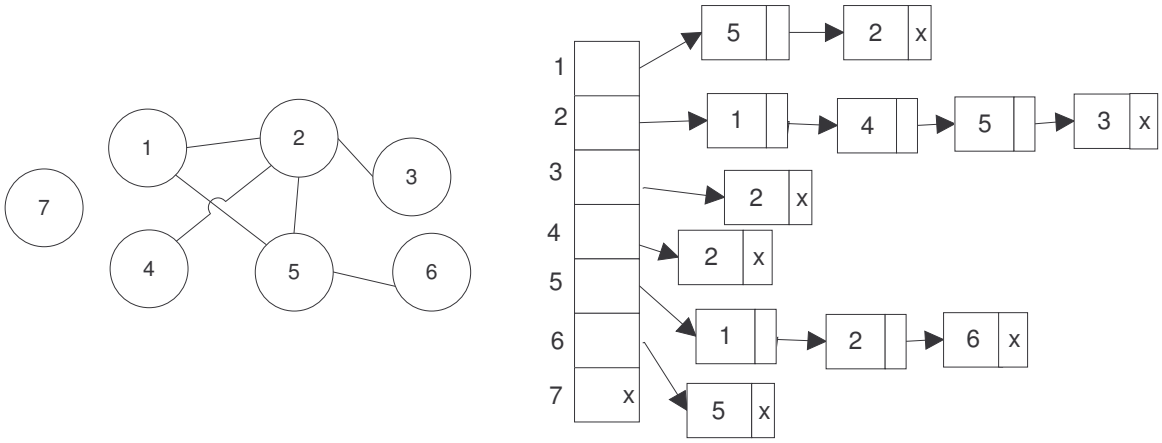
- Crear el grafo (`crea`) es $\theta(n^2)$ siempre que el lenguaje de programación necesite inicializar el espacio reservado para almacenar la matriz. Si no necesita ni inicializar, ni reservar memoria, el coste se puede considerar $\theta(1)$.
- Adyacentes a un vértice (`ady`) y grado de un vértice (`grado`) son θ(n) porque sólo necesitan recorrer la fila correspondiente.
- Añadir una arista (`añ-a`), existe vértice (`ex-v`), existe arista (`ex-a`), borrar una arista (`borra-a`) y valor de la etiqueta de una arista (`valor`) son $\theta(1)$ porque sólo necesitan acceder a una posición de la matriz.
- Borrar o añadir un vértice (`borra-v, añ-v`) son también $\theta(n)$ si lo único que hay que hacer es borrar/inicializar todas las aristas que inciden en el vértice (recorrer la fila y la columna correspondiente al vértice).

En un grafo dirigido distinguimos entre los vértices sucesores y los vértices predecesores de uno dado. Las dos operaciones correspondientes (`suc` y `pred`) tienen coste $\theta(n)$ y corresponden a recorrer la fila y la columna, respectivamente.

Las matrices de adyacencia ofrecen un buen coste espacial y temporal para las operaciones habituales cuando el grafo es denso.

2.2. **Listas y multilistas de adyacencia**

En el caso de las listas de adyacencia, la estructura que se emplea para implementar un grafo G=<V,E> con n =|V|, es un vector L[1..n] tal que L[i], con $1 \leq i \leq n$, es una lista formada por los identificadores de los vértices que son adyacentes al vértice con identificador i. Si el grafo es dirigido, la lista está formada por los identificadores de los sucesores del vértice i.



Un grafo y su implementación usando listas de adyacencia. Cada arista se ha representado 2 veces, una en cada sentido.

El espacio ocupado es de orden $\theta(n+e)$, con e=|E|.

Examinando el coste temporal de algunas operaciones básicas, vemos que:

- Crear la estructura (`crea`) es $\theta(n)$ si el lenguaje de programación necesita reservar e inicializar el espacio ocupado por el vector de n posiciones. Si no es así, el coste es $\theta(1)$.

- Añadir vértice (`añ-v`) y existe vértice (`ex-v`) son $\theta(1)$ siempre y cuando no cueste nada aumentar el tamaño del vector para añadir el nuevo vértice.

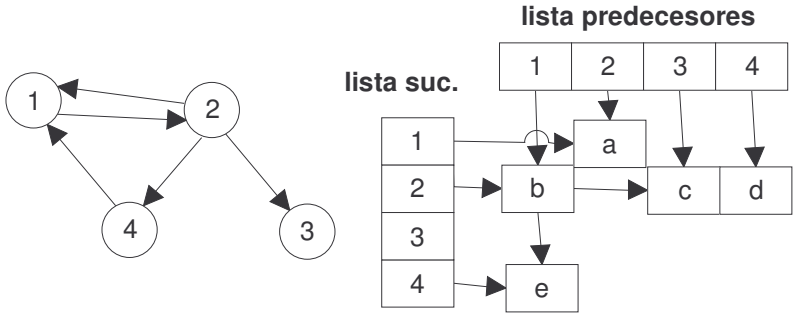
- Añadir arista (`añ-a`), necesita comprobar que la arista que se añade no exista previamente y luego debe añadirla si es el caso. Por tanto el coste de añadir arista depende del de existe arista (`ex-a`) que es $\theta(n)$ en el caso peor ya que tiene que recorrer la lista asociada al vértice. Esta lista puede llegar a tener n-1 elementos. Si procede, insertar una arista en una lista de adyacencia se puede hacer en $\theta(1)$.

- Borrar arista (`borrar-a`) tiene el mismo coste en el caso peor que `añ-a`.

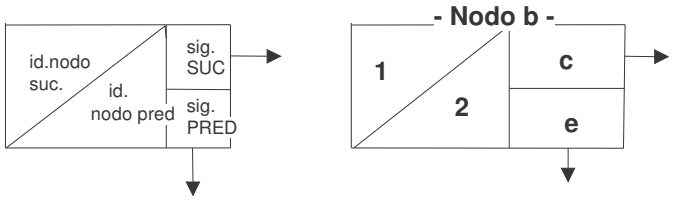
- Borrar vértice (borrar-v) implica borrar completamente la lista que cuelga del vértice a borrar y eliminar todas las apariciones de ese vértice en cualquiera de las otras listas de adyacencia. El coste de la operación es $\theta(n+e)$.
- Para obtener los sucesores (suc) de un vértice en un grafo dirigido o los adyacentes (ady) en un grafo no dirigido, basta con recorrer la lista asociada a ese vértice. Coste en el caso peor $\theta(n)$. Sin embargo, para obtener los predecesores (pred) hay que recorrer toda la estructura para ver en qué listas aparece el vértice del que se están buscando sus predecesores, por tanto $\theta(n+e)$.

Una implementación de la lista de aristas en un AVL (árbol binario equilibrado) permite reducir el coste de la operación ex-a en el caso peor a $\theta(\log n)$ lo que afecta al coste de añ-a que pasa a ser $\theta(\log n)$, borrar-v que pasa a ser $\theta(n+n\cdot\log n)= \theta(n\cdot\log n)$ y pred que también queda $\theta(n\cdot\log n)$.

La implementación de un grafo usando multilistas sólo tiene sentido para grafos dirigidos. Su objetivo es mejorar el coste de la operación pred que pasa a tener coste $\theta(n)$ en lugar del coste $\theta(n+e)$ que tiene con listas de adyacencia. También se reduce a $\theta(n)$ el coste de borrar-v.



Grafo dirigido y su implementación usando multilistas.



Estructura de los nodos de la multilista.

Es conveniente usar listas de adyacencia para implementar el grafo cuando es poco denso y, sobre todo, cuando es necesario recorrerlo completamente.

2.3. **El problema de la celebridad**

Problema: Dado un grafo dirigido G=<V,E> implementado en matriz de adyacencia, determinar en tiempo O(*n*) si existe algún vértice tal que su grado de entrada es *n*-1 y su grado de salida es cero. Si existe, ese vértice se denomina pozo, sumidero o celebridad. Como siempre *n*=|V| .

Se supone que los vértices están identificados por un natural entre 1 y *n* y que la matriz es de booleanos con la interpretación habitual. Esta es la especificación de la función a diseñar.

```
función CELEBRIDAD (M es matriz[1..n,1..n] de bool)
                    dev (b es bool, v es vértice)
```

{*Pre:* CIERTO}

{*Post:* $b \Rightarrow (1 \le v \le n) \wedge (\forall i:(1 \le i \le n) \wedge (i \ne v):(M[i,v]=CIERTO) \wedge (M[v,i]=FALSO))$

$\wedge \neg b \Rightarrow \neg(\exists j:1 \le j \le n:(\forall i:(1 \le i \le n) \wedge (i \ne j):(M[i,j]=CIERTO) \wedge (M[j,i]=FALSO)))$}

De la especificación se deduce que en el caso peor hay que recorrer toda la matriz para determinar que no existe celebridad. Esta solución trivial tiene un coste de $\theta(n^2)$ que es más cara que lo que nos piden.

Se puede intentar el siguiente planteamiento recursivo para reducir el coste:

- BASE INDUCCION : Si el grafo tiene dos nodos, ¿cuántas consultas a la matriz hay que realizar para averiguar si existe celebridad? Es evidente que dos consultas son suficientes.
- HIPOTESIS INDUCCION: Supongamos que tenemos la solución para un grafo con *n*-1 nodos.
- PASO INDUCTIVO: ¿cómo se resuelve para *n* nodos? Se puede producir una de las tres situaciones siguientes:
  - La celebridad es uno de los *n*-1 nodos ya explorados. En ese caso, y para determinar si es definitivamente la celebridad, hay que efectuar dos consultas para ver qué relación existe entre el nodo *n* y la celebridad del grafo con *n*-1 nodos (ha de pasar que no existe arista de la celebridad a *n* pero sí debe existir arista del nodo *n* a la celebridad).
  - La celebridad es el nodo *n*. En ese caso hay que efectuar 2·(*n*-1) consultas para averiguar si sale arista desde los *n-1* nodos del grafo anterior hasta *n* y

no sale ninguna arista de *n* hacia ellos. De este modo se averigua si *n* es la celebridad

- El grafo no tiene celebridad.

Para *n* nodos, y en el caso peor, son necesarias 2·(*n*-1) consultas. Para el paso *n-1* serían necesarias 2·(*n*-2), ..., y así hasta el paso *n*=2 en que se necesitarían 2·(2-1)=2 consultas. Calculando el número total de consultas se obtiene *n*·(*n*-1). ¡Continúa siendo una solución de coste cuadrático!

La forma correcta de resolver el problema consiste en aplicar la técnica denominada *búsqueda por eliminación*. Esta búsqueda aprovecha el hecho de que en cualquier grafo dirigido habrá como mínimo *n*-1 vértices no célebres mientras que celebridad, si hay, sólo habrá una. Es mucho más frecuente encontrar un vértice no célebre que uno célebre. Dado un par de nodos cualesquiera ¿Cuántas consultas hay que efectuar para averiguar si alguno de ellos NO es celebridad? Sólo una pregunta y ya se puede descartar a uno de los dos como candidato a celebridad.

Formalizando este razonamiento, sean *i* y *j* dos nodos cualesquiera:
- si existe arista del nodo *i* al nodo *j* entonces *i* no es celebridad y se puede descartar (el grado de salida del vértice celebridad ha de ser cero).
- si NO existe arista del nodo *i* al nodo *j* entonces *j* no es celebridad y se puede descartar (el grado de entrada del vértice celebridad ha de ser *n*-1).

Dados los *n* nodos del grafo, se necesitan *n*-1 consultas para obtener el nodo candidato a celebridad. Luego hay que comprobar que efectivamente lo es y, como ya se ha mencionado anteriormente, son necesarias 2·(*n*-1) consultas. En total hay que efectuar (*n*-1)+2·(*n*-1) consultas para resolver el problema, lo que corresponde a un tiempo O(*n*) que coincide con lo que pedía el enunciado del problema.

Un algoritmo que implementa esta búsqueda es el que viene a continuación.

```
función CELEBRIDAD(M es matriz[1..n,1..n] de bool) dev (b es bool; v es vértice)
{Pre: CIERTO}
        i:=1; j:=2; p:=3;
        {i y j son los nodos a comparar y p el valor del que se descarte}
        b:= FALSO;
```

```
mientras (p ≤ n+1) hacer
   si (M[i,j]=CIERTO) entonces i:=p        /* se descarta i */
                      sino     j:= p       /* se descarta j */
   fsi
   p:= p+1;
fmientras
```

{ p=n+2 ∧ ((i= n+1 => candidato el j) ∨ (j=n+1 => candidato el i))}

```
si (i=n+1)   entonces v:=j
             sino v:=i
fsi
```

/* Aquí vendría el bucle para comprobar si el candidato p es o no la celebridad. Necesitará efectuar, como máximo, 2·(n-1) consultas. Se deja como ejercicio para el lector */

```
...
dev (b,v)
```

{*Post:* b ⇒ (1≤v≤n) ∧ (∀i : (1≤i≤n) ∧ (i≠v) : (M[i,v]=CIERTO) ∧ (M[v,i]=FALSO))

∧ ¬b ⇒ ¬ (∃j : 1≤j≤n : (∀i : (1≤i≤n) ∧ (i≠j) : (M[i,j]=CIERTO) ∧ (M[j,i]=FALSO)))}

```
ffunción
```

3. ALGORITMOS SOBRE GRAFOS

Este apartado está dedicado básicamente a describir dos algoritmos de recorrido sistemático de un grafo: el recorrido en profundidad y el recorrido en anchura. Otros algoritmos sobre grafos muy conocidos (Prim, Kruskal, Dijkstra, Floyd, etc.) se verán en temas posteriores.

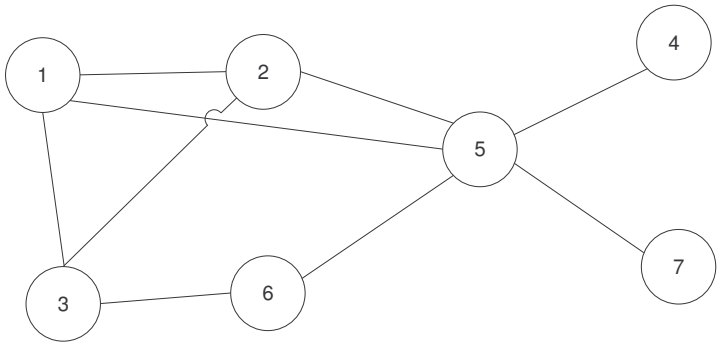
Para facilitar la escritura de los algoritmos que se van a presentar, vamos a considerar que un vértice es una tupla y definiremos los campos que necesitemos en cada uno de los casos. Un campo de los vértices, y que aparecerá en casi todos los algoritmos, será `visitado` que tendrá como valores posibles `'no_visto'` y `'visto'` y que sirven para describir el estado del vértice durante el recorrido. Es evidente que a la hora de implementar los algoritmos habrá que vigilar cómo se hace el paso de parámetros para que los costes sean los que aquí se dan.

3.1. **Recorrido en profundidad**

El algoritmo de recorrido en profundidad, en inglés *depth-first search* y que denotaremos DFS para abreviar, permite efectuar un recorrido sistemático del grafo. El resultado del recorrido puede materializarse en una secuencia que contendrá todos los vértices del grafo. Los vértices aparecen en ella en el orden en que han sido alcanzados (o visitados) por primera vez por el recorrido. El orden de visita es tal que los vértices se recorren según el orden 'primero en profundidad', lo que significa que dado un vértice *v* que no haya sido visitado, la primera vez que el DFS lo alcanza, DFS lo visita y, a continuación, aplica DFS recursivamente sobre cada uno de los adyacentes/sucesores de *v* que aún no hayan sido visitados. Para poner en marcha el recorrido se elige un vértice cualquiera como punto de partida y el algoritmo acaba cuando todos los vértices han sido visitados. Se puede establecer un claro paralelismo entre el recorrido en preorden de un árbol y el DFS sobre un grafo.

El orden de recorrido de los vértices del grafo que produce el DFS no es único, es decir, para un mismo grafo se pueden obtener distintas secuencias de vértices de acuerdo con el orden de visita. Todo depende de en qué vértice comienza el recorrido y de en qué orden se van tomando los adyacentes de un vértice dado.
La siguiente figura ilustra algunas de las diferentes secuencias que puede producir el DFS sobre un grafo no dirigido: El recorrido 1 corresponde al de comenzar por el vértice 1 y recorrer los adyacentes en el orden que marca su etiqueta. El recorrido 2 comienza por el vértice 5 y los adyacentes también se recorren en el orden que

marca su etiqueta. El recorrido 3 comienza por el vértice 7 y elige cómo siguiente vértice a visitar el que tiene la numeración más alta.



Recorrido 1: {1, 2, 3, 6, 5, 4, 7}
Recorrido 2: {5, 2, 1, 3, 6, 4, 7}
Recorrido 3: {7, 5, 6, 3, 2, 1, 4}

Si el grafo es conexo, un único DFS permite visitar todos los vértices y todas las aristas del grafo.

En el algoritmo de recorrido en profundidad para grafos NO DIRIGIDOS que se presenta a continuación a cada vértice se le asocia un valor inicial, `'no_visto'`, para indicar que el recorrido aún no ha pasado por él. En el momento en que el recorrido alcanza un vértice con ese valor, lo modifica poniéndolo a `'visto'`. Significa que se ha llegado a ese vértice por uno, el primero explorado, de los caminos que lo alcanzan y que se viene de un vértice que también ha sido ya visitado (excepto para el vértice inicial). Nunca más se alcanzará ese vértice por el mismo camino (nunca se volverá a entrar por esa arista), aunque sí se podrá entrar por otras aristas distintas pero el vértice ya estará marcado a `'visto'`.

```
acción REC_PROF(g es grafo)
{Pre: CIERTO}
        Para cada v∈V hacer  v.visitado:= 'no_visto' fpara
        Para cada v∈V hacer
                si no_visto(g,v) entonces  g:= REC_PROF_1(g,v);
                fsi
        fpara
{Post: Todos los vértices de g han sido marcados a visto en el orden que sigue el recorrido en profundidad de g}
faccción
```

```
función  REC_PROF_1(g es grafo; u es vértice) dev (g es grafo)
{Pre : (u∈ V) ∧ ¬Visto(g,u)  ∧ (g=g')}
        u.visitado:='visto';
        TRATAR(u);                          /* PREWORK */
        Para cada w ∈ ady(g,u) hacer
                si no_visto(g,w) entonces  g:= REC_PROF_1(g,w);
                fsi
                TRATAR(u,w);                /* POSTWORK */
        fpara
{Post : Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados
exclusivamente por vértices que no estaban vistos en g',  según el orden de visita del recorrido en profundidad }
        dev  g;
ffunción
```

La función auxiliar `REC_PROF_1` es la que realmente hace el recorrido en profundidad. En ella aparecen dos operaciones `TRATAR(u)` y `TRATAR(u,w)` que pueden corresponder a acciones vacías, como sucede en el caso del `DFS` general, o pueden corresponder a acciones relevantes, como se verá en algún algoritmo posterior. En general, la acción denominada *prework* designa al trabajo previo sobre el vértice visitado, y la denominada *postwork* corresponde al trabajo posterior sobre la última arista tratada (o sobre el mismo vértice sobre el que se hace el *prework*).

Vamos a analizar el coste de `REC_PROF` suponiendo que *prework* y *postwork* tienen coste constante. Si el grafo está implementado sobre una matriz de adyacencia el coste es $\theta(n^2)$ ya que todos los vértices (los n) se marcan a `'visto'` una sola vez (el bucle exterior de `REC_PROF` así lo garantiza), y cada vez que se marca uno de ellos se consultan todos sus adyacentes (coste $\theta(n)$) para decidir si hay que lanzar o no llamada recursiva. Si el grafo está implementado con listas de adyacencia, lo único que cambia es que la consulta de los adyacentes tiene coste $\theta(n)$ pero sólo en el caso peor y el trabajo que hacen todas las llamadas recursivas corresponde al recorrido de todas las aristas del grafo (si el grafo es dirigido por cada arista sólo se pasa una vez pero si es no dirigido se pasa dos veces, una en cada sentido). En total coste es $\theta(n+e)$.

Los apartados siguientes se dedican a presentar algunas de las múltiples aplicaciones del `DFS`: determinar si un grafo es conexo, numerar los vértices según

el orden del recorrido, determinar la existencia de ciclos, etc. La mayoría de ellas se implementan a base de modificar, levemente, el algoritmo de recorrido en profundidad, lo que da una idea de su potencia.

3.1.1. *Conectividad*

Se trata de determinar si un grafo no dirigido es conexo o no lo es. Podemos usar la propiedad de que si un grafo es conexo es porque existe camino entre todo par de vértices o, lo que es lo mismo, a partir de cualquier vértice es posible alcanzar a todos los demás. Se necesita un algoritmo que recorra todo el grafo y que permita averiguar qué vértices son alcanzables a partir de uno dado. El DFS es el adecuado. El bucle de la función externa, que aquí denominaremos COMPO_CONEXAS, examina todos los vértices y si alguno no ha sido visitado comienza un recorrido en profundidad a partir de él. Las llamadas desde COMPO_CONEXAS a CONEX_1 indican el comienzo de la visita de una nueva componente conexa y se puede asegurar que cuando se regresa de la llamada recursiva no hay más vértices que formen parte de esa misma componente conexa. En la implementación del algoritmo se ha utilizado un natural, nc, que se asocia a cada vértice y que indica a qué número de componente conexa pertenece.

```
función COMPO_CONEXAS(g es grafo) dev (g es grafo; nc es nat)
```
{*Pre:* g es un grafo no dirigido }
```
        Para cada v∈V hacer  v.visitado:= 'no_visto'; v.ncc:= 0 fpara
        nc:= 0;
        Para cada v∈V hacer
                si no_visto(g,v) entonces  nc:= nc+1; g:= CONEX_1(g, v, nc);
                fsi
        fpara
```
{*Post:* el grafo ha sido modificado según figura en la Post de CONEX_1 ∧ *nc* contiene el número de componentes conexas que tiene el grafo g}
```
        dev (g,nc);
ffunción
```

```
función CONEX_1(g es grafo; u es vértice; num_compo es nat) dev (g es grafo)
```
{*Pre:* (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g') ∧ (num_compo = nº de componente conexa que se está recorriendo) }
```
        u.visitado:= 'visto';  u.ncc:= num_compo;
```
/* Este es el PREWORK, anotar el número de componente conexa a la que pertenece el vértice.*/

```
        Para cada w ∈ ady(g,u) hacer
                si no_visto(g,w) entonces  g:= CONEX_1(g,w,num_compo);
                fsi
        fpara
```

{*Post:* Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad. Además a todos los vértices visitados se les ha añadido una información que indica a qué nº de componente conexa pertenecen}

```
        dev g;
ffunción
```

El coste de este algoritmo es idéntico al de `REC_PROF`.

3.1.2. *Numerar vértices*

Aprovechando el recorrido en profundidad se puede asociar a cada vértice un valor natural que indicará el orden en que han sido visitados (han pasado a `'visto'`). A esta asociación se la conoce como la numeración en *el orden del recorrido*. También se puede asociar a cada vértice un natural, no en el momento en que pasa a `'visto'`, sino en el momento en que el recorrido se da cuenta de que todos sus adyacentes ya han sido visitados. A esta otra numeración se la conoce como la del *orden inverso del recorrido*.

En el algoritmo que se presenta a continuación hay dos valores, de tipo natural, que se almacenan en cada vértice: `num-dfs`, que contiene la numeración en el orden del recorrido, y `num-invdfs` que contiene la numeración en orden inverso. Con una pequeña modificación se puede conseguir que el algoritmo funcione para grafos dirigidos.

```
función NUMERAR_VERTICES(g es grafo) dev (g es grafo)
```

{*Pre :* g es un grafo no dirigido}

```
        Para cada v∈V hacer
            v.visitado:= 'no_visto'; v.num-dfs:= 0; v.num-invdfs:= 0;
        fpara
        ndfs:= 0; ninv:= 0 ;
        Para cada v∈V hacer
           si no_visto(g,v) entonces  <g, ndfs, ninv>:= NV_1(g, v, ndfs, ninv);
```

```
            fsi
        fpara
```

{*Post :* los vértices del grafo han sido marcados según se indica en la Post de NV_1}

```
        dev g;
ffunción
```

```
función NV_1(g es grafo; u es vértice; nd, ni es nat)
          dev (g es grafo; nd, ni es nat)
```

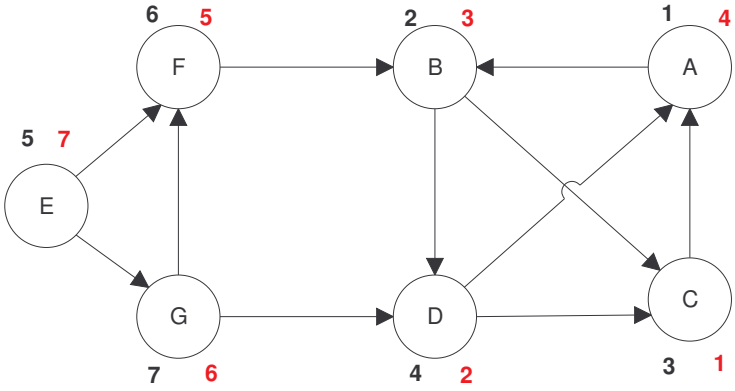
{*Pre:* (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g') ∧ (*nd* es el número asociado al último vértice que ha pasado a 'visto' en el recorrido) ∧ (*ni* es el número asociado al último vértice que en el recorrido se ha dado cuenta de que tenía todos sus adyacentes visitados) }

```
        u.visitado:= 'visto';  nd:= nd+1;
        u.num-dfs:= nd;                          /* PREWORK sobre el vértice */
        Para cada w ∈ ady(g,u) hacer
            si no_visto(g,w) entonces
                    <g, nd, ni> := NV_1(g, w, nd, ni);
            fsi
        fpara
        ni:= ni+1;
        u.num-invdfs:= ni;                       /* POSTWORK sobre el vértice */
```

{*Post:* visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad ∧ *u.num-invdfs* es la numeración en orden inverso del vértice u que es el último que ha conseguido tener todos sus adyacentes a 'visto' ∧ *nd* es la numeración en el orden de recorrido del último vértice de g que ha pasado a visto}

```
        dev  (g, nd, ni);
ffunción
```

La siguiente figura muestra un grafo dirigido que ha sido recorrido en profundidad y numerado en el orden del recorrido (valor que aparece a la izquierda, en negro) y en el orden inverso (valor que aparece a la derecha, en rojo). Se ha comenzado el recorrido DFS por el vértice con identificador *A* y el siguiente se ha elegido según el orden alfabético.



3.1.3. *Árbol asociado al recorrido en profundidad*

Al mismo tiempo que se efectúa el recorrido en profundidad de un grafo se puede obtener lo que se denomina el *árbol asociado al recorrido en profundidad*. En realidad no siempre se obtiene un árbol sino que depende del grafo de partida. Así, un grafo no dirigido y conexo produce un árbol, un grafo no dirigido y no conexo produce un bosque, un grafo dirigido y fuertemente conexo produce un árbol, y un grafo dirigido y no fuertemente conexo produce un bosque (que puede tener un único árbol). El árbol se caracteriza por contener todos los vértices del grafo de partida junto con todas aquellas aristas que durante el recorrido del grafo cumplen la condición de que uno de sus extremos es un vértice marcado a `'visto'` y el otro extremo es un vértice marcado a `'no_visto'`.

```
función ARBOL_DFS(g es grafo) dev (B es bosque)
```

{*Pre:* g es un grafo no dirigido}

```
        Para cada v∈V hacer  v.visitado:= 'no_visto' fpara
        B := bosque_vacio;
        Para cada v∈V hacer
                si no_visto(g, v) entonces
                        T:= arbol_vacio;
                        <g, T>:= TDFS_1(g, v, T);
                        B:= añadir_arbol(B, T);
                fsi
        fpara
```

{*Post :* los vértices del grafo han sido visitados según el orden de recorrido en profundidad ∧ *B* es el bosque de árboles DFS que ha producido este recorrido}

```
        dev B;
ffunción
```

```
función TDFS_1(g es grafo; u es vértice; T es árbol) dev (g es grafo; T es árbol)
```

{*Pre :* (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g') ∧ (T es el árbol asociado al recorrido en profundidad de g hasta este momento y contiene todos los vértices marcados a 'visto') ∧ (T=T') }

```
        u.visitado:= 'visto';
        T:= añadir_vértice(T, u);                    /* PREWORK */
        para cada w ∈ ady(g,u) hacer
                si no_visto(g, w) entonces
                        <g, T>:= TDFS_1(g, w, T);
                        T:= añadir_arista(T, u, w);
                fsi;
```

/* el POSTWORK añade a T aristas que cumplen la condición de que uno de los extremos ya está marcado a 'visto' y el otro no (eso sucedía antes de la llamada recursiva) */

```
        fpara
```

{*Post:* Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad ∧ T=T'∪{aristas visitadas después de hacer el recorrido en profundidad a partir del vértice u} }

```
        dev  (g, T);
ffunción
```

Conviene caracterizar formalmente el árbol asociado al DFS, lo denotaremos por T_{DFS}, y para ello resultaran útiles las siguientes definiciones y lema.

Definición: Un vértice *v* es un antecesor del vértice *w* en un árbol T con raíz *r*, si *v* está en el único camino de *r* a *w* en T.

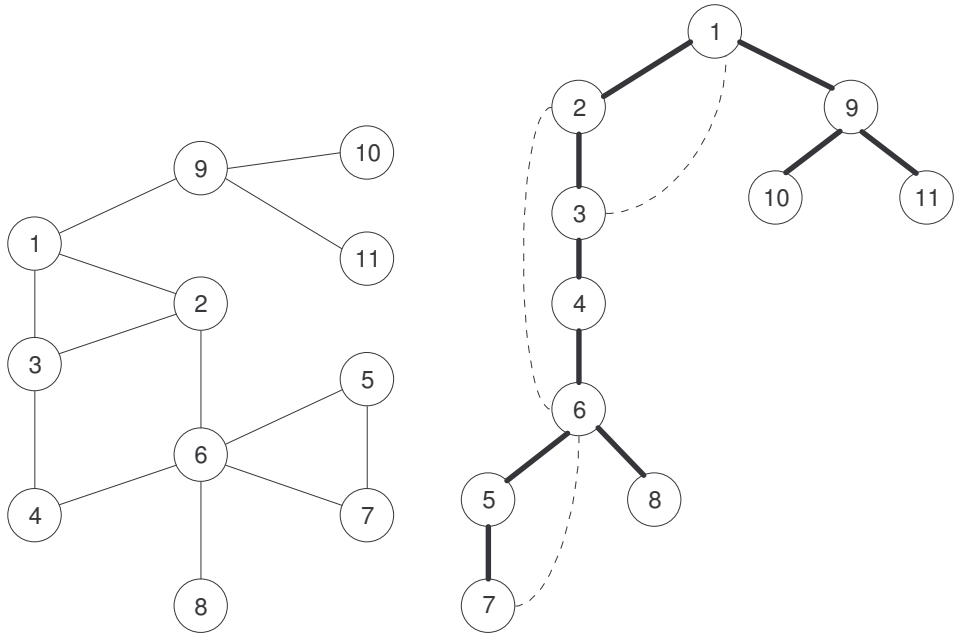
Definición: Si *v* es un antecesor de *w*, entonces *w* es un descendiente de *v*.

Lema: Sea G=<V,E> un grafo conexo no dirigido, sea T_{DFS}=<V,F> el árbol asociado al recorrido en profundidad de G. Entonces toda arista *e*, *e*∈ E, o bien aparece en el T_{DFS}, es decir, *e*∈ F, o bien no aparece en el T_{DFS} y conecta dos vértices de G uno de los cuales es antecesor del otro en T_{DFS}.

Una vez fijado un T_{DFS}, el lema permite clasificar las aristas de G en dos grupos:

- GRUPO 1: las aristas de G que aparecen en el T_{DFS} que son las *tree edges* o aristas del árbol. Estas aristas conectan padres con hijos en el árbol.
- GRUPO 2: las aristas de G que no aparecen en el T_{DFS} y que son las *back edges* o aristas de retroceso o de antecesores. Estas aristas conectan descendientes con antecesores en el árbol.

En la siguiente figura se muestra un grafo no dirigido y el T_{DFS} que se ha obtenido iniciando el recorrido de G en el vértice 1 y decidiendo el siguiente en orden numérico. Las aristas marcadas en negrita corresponden a las que forman parte del T_{DFS}, *tree edges*, mientras que las restantes son las *back edges*, es decir, aristas de G que no aparecen en el árbol y que conectan descendientes con antecesores.



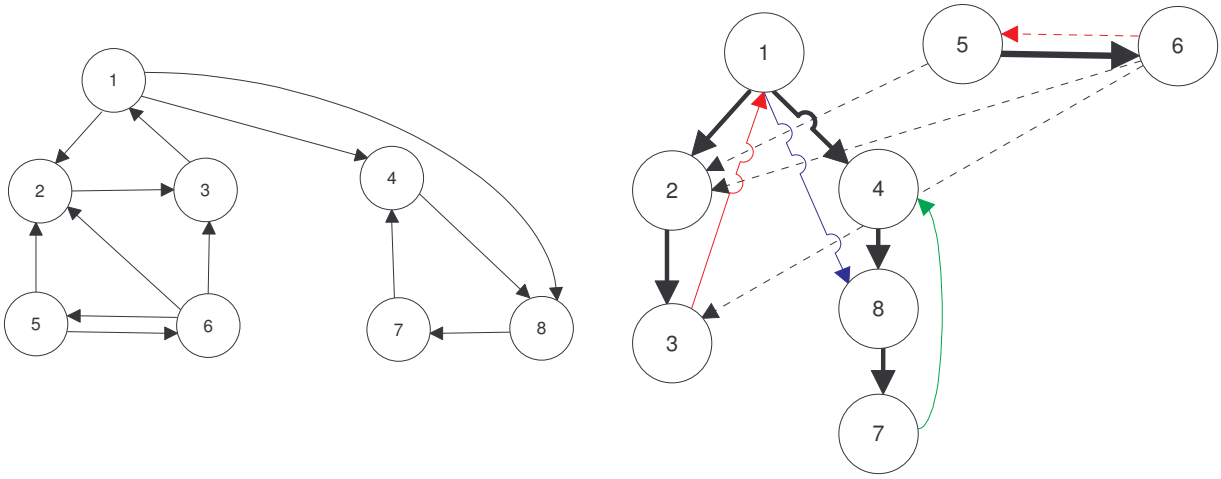
Grafo de entrada y árbol asociado al DFS (aristas tree en negrita y aristas back en línea discontínua)

En un grafo DIRIGIDO, G=<V,E>, sus aristas se pueden clasificar en cuatro tipos en función del T_{DFS} que produce el recorrido en profundidad. Así se tienen:

- *tree edges* y *back edges* con la misma definición que se ha dado para los grafos no dirigidos.
- *forward edges*, o aristas descendientes, que conectan antecesores con descendientes en el árbol.
- *cross edges*, o aristas cruzadas, que conectan vértices no relacionados en el T_{DFS}. Estas aristas siempre van de derecha a izquierda en el árbol (o de izquierda a derecha, todo depende de cómo se dibuje).

La numeración en el orden del recorrido en profundidad de los vértices del grafo permite determinar el tipo de las aristas del grafo o las relaciones en el T_{DFS} entre los vértices de una arista. El siguiente lema es una muestra de ello.

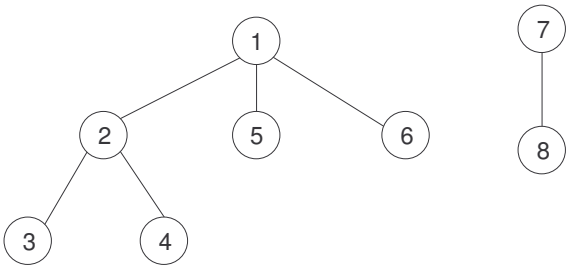
Lema: Sea G=<V,E> un grafo dirigido y sea T_{DFS}=<V,F> el árbol asociado al recorrido en profundidad de G. Si (*v,w*)∈E y `v.num-dfs < w.num-dfs` entonces *w* es un descendiente de *v* en el árbol T_{DFS}.



Grafo dirigido y árbol asociado al recorrido en profundidad (en negrita aristas tree, en rojo aristas back, en verde aristas forward y en línea discontínua aristas cross)

En la figura anterior se muestra un grafo dirigido y el árbol asociado al recorrido en profundidad que se ha iniciado en el vértice 1 y se ha aplicado orden numérico para elegir el siguiente vértice. Las aristas en negrita son las *tree edges*, es decir, las que forman el T_{DFS}.

Observando únicamente el árbol asociado al recorrido en profundidad se pueden determinar algunas de las características del grafo de partida. Por ejemplo, dado el siguiente árbol producido por el DFS sobre un grafo G:



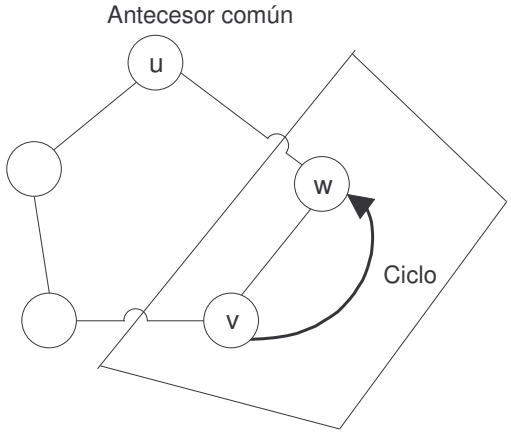
se puede deducir que el grafo G no es conexo ni dirigido y que el vértice 5 no es adyacente ni al vértice 3 ni al 4 y que los vértices 3 y 4 no son adyacentes entre sí, etc.

3.1.4. *Test de ciclicidad*

Una de las utilidades del árbol asociado al recorrido en profundidad es que permite averiguar si el grafo, dirigido o no, contiene ciclos. El siguiente lema, para grafos dirigidos, así lo indica:

Lema: Sea G=<V,E> un grafo dirigido y T_{DFS} el árbol del recorrido en profundidad de G. G contiene un ciclo dirigido si y sólo si G contiene una arista de retroceso.

Prueba: Sea C un ciclo en G y sea *v* el vértice de C con la numeración más baja en el orden del recorrido, es decir, `v.num-dfs` es menor que el `num-dfs` de cualquier otro vértice que forma parte de C. Sea (*w,v*) una de las aristas de C. ¿qué clase de arista es ésta? Forzosamente `w.num-dfs` será mayor que `v.num-dfs` y esto hace que se descarte que la arista sea una *tree edge* o una *forward edge*. Podría tratarse solamente de una *back edge* o de una *cross edge*. Si fuera una arista cruzada significaría que *v* y *w* tienen un antecesor común *u*, con `u.num-dfs` menor que `v.num-dfs`, que es la única forma de conexión entre ellos, además de a través de la arista (*w,v*). El gráfico siguiente ilustra esta situación.



Sin embargo, esto no es posible ya que *v* y *w* comparten un ciclo y, por tanto, existe camino entre *v* y *w* pero ¡sólo atravesando vértices con una numeración mayor que la de *v*! La única posibilidad que nos queda es que (*w,v*) sea un *back edge*, que implica que si hay arista de retroceso es que hay ciclo.

El algoritmo que se presenta a continuación detecta la presencia de una arista de retroceso. El método que sigue es mantener, para el camino que va desde la raíz al vértice actual, qué vértices lo componen. Si un vértice aparece más de una vez en ese camino es que hay ciclo. Esta versión tiene el mismo coste que el algoritmo `REC_PROF`.

```
función CICLOS(g es grafo) dev (b es bool)
{Pre : g es un grafo dirigido }
        Para cada v∈V hacer
                v.visitado:= 'no_visto';
                v.en-camino:= FALSO;
                /* no hay ningún vértice en el camino de la raíz al vértice actual */
        fpara
```

```
        b:= FALSO;                          /* inicialmente no hay ciclos */
        Para cada v∈V hacer
                si no_visto(g, v) entonces
                        <g, b1> := CICLOS_1(g, v);     b:= b1 ∨ b;
                fsi;
        fpara
```

{*Post :* los vértices del grafo han sido recorridos según el orden de recorrido en profundidad ∧ *b* = Cíclico (g)}

```
        dev  b;
ffunción

función CICLOS_1(g es grafo; u es vértice) dev (g es grafo; b se bool)
```

{*Pre :* (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g')}

```
        u.visitado:= 'visto';  u.en-camino:= CIERTO;
```

/* PREWORK :Se anota que se ha visitado u y que éste se encuentra en el camino de la raíz a él mismo*/

```
        b:= FALSO;
        Para cada w ∈ suc(g,u) hacer
           si visto(g,w) entonces
                 si w.en-camino entonces b1:= CIERTO;
                          /* Ciclo!!, se recorre la arista (u,w) pero ya existía camino de w a u */
                 fsi
           sino <g, b1>:= CICLOS_1(g, w);
           fsi
           b:= b1 ∨ b
        fpara
```

/* ya se han recorrido todos los sucesores de u y se abandona el camino actual desde la raíz (se va 'desmontando' el camino y u ya no forma parte de él). La siguiente asignación corresponde al POSTWORK */

```
        u.en-camino := FALSO
```

{*Post:* Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad ∧ *b* dice si en el conjunto de caminos que tienen en común desde la raíz hasta u y luego se completan con la descendencia de u, hay ciclos.}

```
        dev (g, b);
ffunción
```

Los grafos dirigidos y acíclicos, *directed acyclic graphs*, DAG, se utilizan en muchos ámbitos. Por ejemplo, el plan de estudios de la facultad es un DAG y las expresiones

aritméticas con subexpresiones comunes que se pueden escribir en cualquier lenguaje de programación también deben ser un DAG.

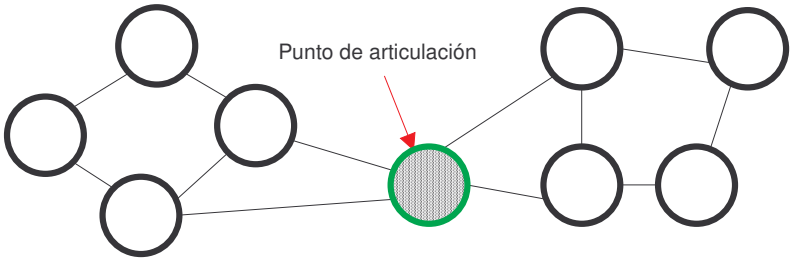
Si en lugar de trabajar con un grafo dirigido se trabajara con uno no dirigido, el algoritmo anterior sería más simple. La existencia de un ciclo se detectaría cuando desde un vértice `'visto'` se alcanza otro que también lo está, excepto si se trata del padre en el recorrido. Para detectar la excepción sería necesario que cada llamada recibiera, además, el vértice que la ha provocado (el padre de la llamada).

Un problema interesante, y que se deja como ejercicio para el lector, es el siguiente: Dado un grafo no dirigido, G=<V,E>, proponer un algoritmo que en tiempo O(*n*), *n*=|V|, determine si existen ciclos o no. Notar que el coste ha de ser independiente de |E|.

3.1.5. *Puntos de articulación*

Sea G=<V,E> un grafo conexo, se dice que el vértice *v*, *v*∈V, es un punto de articulación si el subgrafo G'=<V–{*v*}, E–{todas las aristas de E de la forma (*v*,*w*)}> deja de ser conexo. La detección de puntos de articulación en una red de comunicaciones permite identificar sus puntos débiles. Las dos definiciones que vienen a continuación identifican las características que ha de tener una red para soportar fallos en la conectividad (cae un nodo de la red o falla una conexión).

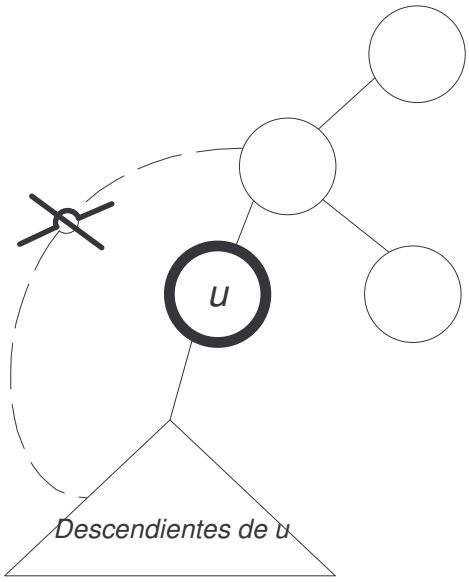
Definición: Un grafo es biconexo si es conexo y no tiene puntos de articulación.
En una red que resulte ser un grafo biconexo aunque falle un nodo, el resto de la red sigue manteniendo la conectividad.
Definición: Un grafo bicoherente es un grafo en el que todo punto de articulación está unido mediante al menos dos aristas con cada una de las componentes del grafo que quedarían al eliminar el vértice punto de articulación.
Una red con estas características sigue funcionando aunque falle una línea de la red (una arista).



Grafo bicoherente

Vamos con las ideas para plantear el algoritmo que calcula los puntos de articulación. Supongamos que tenemos calculado el T_{DFS} del grafo G=<V,E> y que sabemos qué aristas de G son *back*. Sea *r*= raíz(T_{DFS}):

1. *r* es un punto de articulación si tiene más de un hijo en el árbol.
2. $\forall u \in V$, $u \neq r$, *u* es punto de articulación si al eliminarlo del árbol desde ninguno de sus descendientes se puede 'ascender' hasta alguno de los antecesores de *u* en el árbol. Gráficamente



   Para cada vértice *u* hemos de averiguar hasta dónde puede subir (ascender) en el árbol sabiendo que para remontar hay que utilizar aristas *back*.

Refinando la idea y suponiendo que cada vértice 'sabe' hasta donde puede subir (para ello usaremos un vector indexado por número de vértice, `masalto`, que contiene la numeración en el orden del recorrido del vértice más alto al que puede ascender), podemos hacer el siguiente análisis:
Sea *u* un vértice cualquiera de V:

– Si *u* no tiene hijos en el T_{DFS}, entonces *u* NO es punto de articulación.
– Si *u* tiene hijos en el T_{DFS}, sea *x* un hijo de *u* tal que:

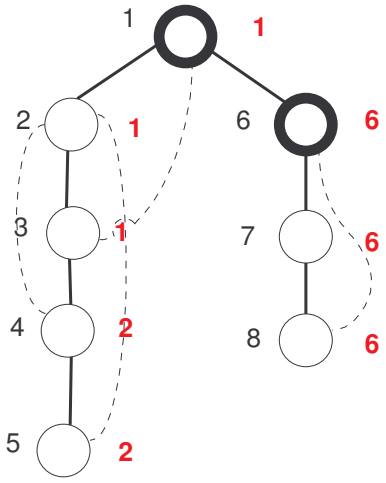
- `masalto[x] < u.num-dfs`, si eliminamos el vértice *u* entonces los vértices del subárbol cuya raíz es *x* NO quedan desconectados del resto del árbol porque partiendo de *x* existe una cadena de aristas de G, que no incluye la arista (*u,*x), y que nos lleva por encima de *u*.
- `masalto[x] ≥ u.num-dfs`, si eliminamos *u* los vértices del subárbol cuya raíz es *x* quedan desconectados del resto del árbol porque

partiendo de *x* no existe una cadena de aristas en G que nos lleve por encima de *u*.

Entonces, *u* es punto de articulación si y sólo si tiene un hijo *x* tal que cumple la condición que `masalto[x] ≥ u.num-dfs`.

El vértice más alto al que se puede ascender a partir de un vértice y fijado un T_{DFS} se calcula de la siguiente forma:

$\forall u \in V$: `masalto[u]` = **MIN** (`u.numdfs`,
`w.numdfs`, $\forall w \in V$ con $(u,w) \in E$ y $(u,w) \notin T_{DFS}$,
`masalto[x]`, $\forall x \in$ hijos(T_{DFS},u))



A la izquierda de cada nodo aparece la numeración en el orden del recorrido, `num-dfs`, y a la derecha en rojo el número del vértice más alto al que se puede remontar, `masalto`. Las aristas continuas son aristas del T_{DFS} y las discontinuas son las aristas *back* del grafo. Los vértices con numeración en el orden de recorrido 1 y 6 son puntos de articulación.

3.1.6. *Componentes fuertemente conexas*

Un grafo dirigido es fuertemente conexo si para todo par de vértices diferentes, *u* y *v*, existe camino de *u* a *v* (*u* ~> v) y existe camino de *v* a *u* (*v* ~> *u*).

Sea G=<V,E> un grafo dirigido, el siguiente algoritmo calcula las componentes fuertemente conexas de G, CFC(G):

1. Recorrido en profundidad de G etiquetando los vértices con la numeración en el orden inverso del recorrido (etiquetas `num-invdfs`).

2. Calcular G', siendo G' el grafo traspuesto de G (el que se obtiene invirtiendo el sentido de todas las aristas de G). Las componentes fuertemente conexas de ambos grafos, G y G', son las mismas.
3. Recorrido en profundidad de G' pero en orden decreciente de las etiquetas `num-invdfs`. Anotar cada uno de los árboles encontrados. Cada uno de ellos es una componente fuertemente conexa de G.

El coste de este algoritmo es $\theta(n+e)$.

Para argumentar la corrección del algoritmo hay que destacar 2 hechos básicos:

1. Si dos vértices pertenecen a la misma componente fuertemente conexa, CFC para abreviar, el camino que los une nunca sale de la CFC.

   Prueba: Por definición, si $a,b \in V$ y ambos están en la misma CFC entonces $a \rightsquigarrow b$ y $b \rightsquigarrow a$. Sea w un vértice en el camino de a a b ($a \rightsquigarrow w \rightsquigarrow b$). Sabemos que existe camino $w \rightsquigarrow b \rightsquigarrow a$ y, por tanto, a, b y w pertenecen a la misma CFC. Generalizando este razonamiento para todos los vértices en el camino de a a b, concluimos que todos ellos pertenecen a la misma CFC.
2. Sea cual sea el recorrido en profundidad realizado, todos los vértices que están en la misma CFC aparecen en el mismo árbol asociado al recorrido en profundidad, T_{DFS}.

   Prueba: Sea r el primer vértice que pasa a `'visto'` de una CFC. Sabemos que existe camino desde r al resto de vértices de la misma CFC. Todos ellos serán descendientes de r y por tanto aparecerán en el mismo T_{DFS}.

Definición: El progenitor de u, $\phi(u)$, es aquel vértice w tal que se puede alcanzar desde u y que tiene el `num-invdfs` más grande de entre todos los alcanzables a partir de u:

$\phi(u)=w$, tal que $u \rightsquigarrow w$ y `w.num-invdfs` es máximo
➔ `u.num-invdfs` $\leq \phi(u)$`.num-invdfs`.

Teorema: En un grafo dirigido G=<V,E>, el $\phi(u)$ de cualquier vértice $u \in V$ es un antecesor de u en el T_{DFS}.

Prueba: Si $u=\phi(u)$ es trivialmente cierto (un vértice es antecesor de él mismo).
Si $u \neq \phi(u)$,

- $\phi(u)$ no puede ser descendiente de u porque entonces `u.num-invdfs` > $\phi(u)$`.num-invdfs`, lo que es imposible por definición.

- No puede ser que no exista relación entre *u* y $\phi(u)$ porque entonces ambos tendrían un antecesor común, por ejemplo *x*, con un `x.num-invdfs` mayor que el de ellos y se contradiría la elección de $\phi(u)$.
- Si ha de existir relación entre ellos, y no es que $\phi(u)$ sea descendiente de *u*, entonces es que $\phi(u)$ es antecesor de *u* en el T_{DFS}.

Corolario: Para cualquier recorrido en profundidad de un grafo dirigido G=<V,E> y para todo vértice $u \in V$, los vértices *u* y $\phi(u)$, su progenitor, pertenecen a la misma CFC.

Prueba: Por definición de progenitor *u* ~> $\phi(u)$ y por el teorema anterior sabemos que $\phi(u)$ ~> *u* ya que $\phi(u)$ es un antecesor de *u* en el T_{DFS}. Luego ambos pertenecen a la misma CFC.

Y ya el último teorema:

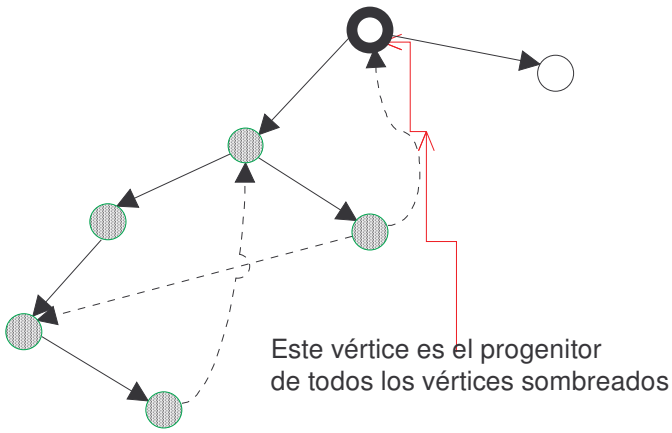
Teorema: Sea G=<V,E> un grafo dirigido, dos vértices *u* y *v* pertenecen a la misma CFC si y solo si tienen el mismo progenitor en el recorrido en profundidad de G.

Prueba:

➔ Supongamos que *u* y *v* pertenecen a la misma CFC. Todos los vértices alcanzables desde *u* también son alcanzables desde *v* y viceversa ya que hay caminos entre ambos en los dos sentidos. Por definición de progenitor se puede concluir que $\phi(u) = \phi(v)$.

➔ Supongamos que $\phi(u) = \phi(v)$ y por el corolario anterior *u* y $\phi(u)$ están en la misma CFC y lo mismo les pasa a *v* y $\phi(v)$. Por tanto, *u* y *v* están en la misma CFC.

Resumiendo, el progenitor es el vértice de la CFC que primero pasa a `'visto'` y el último al que se le asocia la numeración en el orden inverso del recorrido.

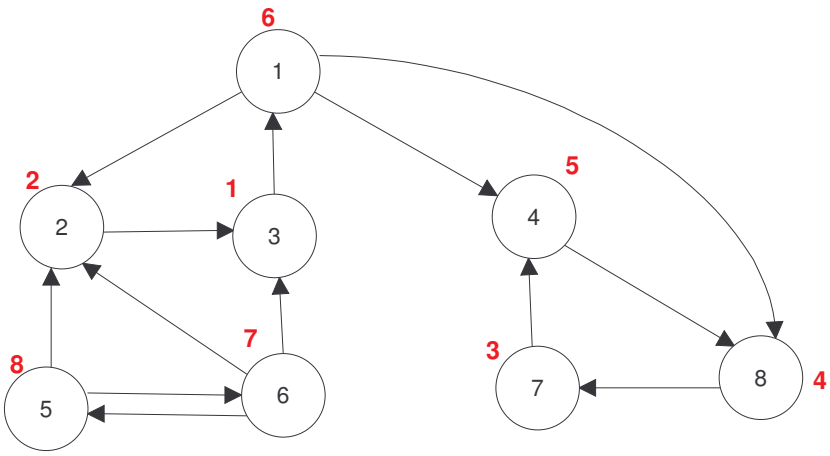


Por eso, una vez calculada la numeración en el orden inverso, invertimos las aristas del grafo y lanzamos el recorrido en profundidad pero comenzando por el vértice

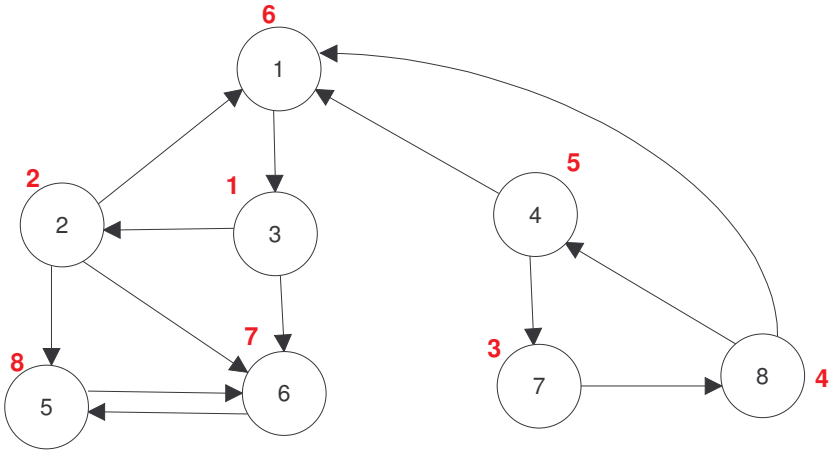
con la mayor numeración en el orden inverso del recorrido sin que sea necesario identificar el progenitor de cada vértice. Después de ese primer recorrido estarán en el mismo árbol todos los vértices de la misma CFC. Repetimos el proceso lanzando un nuevo recorrido en profundidad desde el vértice con la mayor numeración en el orden inverso del recorrido que todavía no hayamos visitado y así sucesivamente. Resolveremos el problema y obtendremos tantos árboles como CFC tenga el grafo.

Un ejemplo para acabar. Sea G=<V,E> el grafo de la figura,

1. Lanzamos un recorrido en profundidad (o los que hagan falta) y asociamos a cada vértice la numeración en el orden inverso que le corresponde en ese recorrido (valor en rojo fuera del nodo):

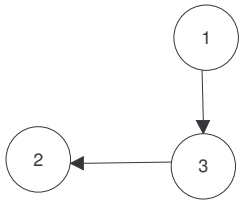


2. Invertimos las aristas de G y obtenemos G':

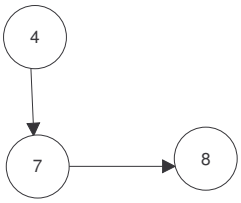


3. Recorrido en profundidad comenzando por el vértice con la mayor numeración en el orden inverso (vértice con etiqueta 5 y numeración 8) que genera el árbol:

4. Recorrido en profundidad comenzando por el vértice con la mayor numeración en el orden inverso (vértice con etiqueta 6 y numeración 6) que genera el árbol:



5. Idem comenzando por el vértice con etiqueta 4 y numeración 5 que genera el árbol:



Y ya tenemos las 3 CFC calculadas.

3.1.7. *Un TAD útil : MFSets*

El tipo de datos denominado *Merge-Find Sets*, MFSets o *Disjoint-Set Data Structure* es el más adecuado para problemas del tipo 'determinar qué elementos de un conjunto pertenecen a la misma clase de equivalencia respecto de la propiedad P'. Una aplicación inmediata sobre grafos es la de determinar las componentes conexas de un grafo no dirigido o averiguar si existe camino entre dos vértices dados.

En general, el objetivo de esta estructura es, dado un conjunto de elementos $CE=\{e_1, e_2, \ldots, e_r\}$, mantener una colección, $S=\{S_1, S_2, \ldots, S_n\}$, de conjuntos disjuntos de los elementos de CE de forma dinámica. Las operaciones habituales son: iniciar, añade, find y merge.
La operación iniciar crea la estructura vacía (colección vacía), añade coloca un conjunto en la colección, find indica en qué conjunto de la colección se encuentra un elemento y merge fusiona dos conjuntos de la colección. Estas dos últimas

operaciones son las más frecuentes durante la vida de la estructura, por eso todas las implementaciones van orientadas a rebajar su coste.
La primera implementación es muy simple. La colección se implementa sobre un vector de *n* posiciones, tantas como elementos haya que clasificar. El índice del vector identifica el elemento y el contenido del vector sobre ese índice contiene la etiqueta del conjunto al que pertenece (la etiqueta coincide con alguno de los elementos del conjunto). Inicialmente cada conjunto contiene un único elemento (se realizan *n* operaciones añade) y el índice y la etiqueta del conjunto coinciden. El siguiente algoritmo es una implementación de la operación merge.

```
función MERGE(mf es vector[1.. n] de nats; a, b es nat) dev (mf es vector[1.. n] de nats )
```

{*Pre:* a y b son las dos etiquetas de los conjuntos que hay que fusionar. En este caso, y como los elementos son naturales, las etiquetas también lo son}

```
        i:= 0;
        mientras (i < n) hacer
                i:= i+1;
                si (mf[i]=b) entonces mf[i]:= a;
                fsi
        fmientras
```

{*Post:* todos los elementos entre 1 y n que pertenecían al conjunto con etiqueta b, ahora pertenecen al conjunto con etiqueta a (se ha fusionado a y b). Los demás elementos siguen en el mismo conjunto en el que estaban}

```
        dev ( mf )
ffunción
```

Para esta implementación, el coste de la operación merge es $\theta(n)$ mientras que el de la operación find es $\theta(1)$.

La segunda implementación es más astuta. Todos los elementos de un conjunto están en un árbol en el que sus elementos tienen puntero al padre. La colección se implementa sobre un vector de tamaño *n* (el número de elementos a clasificar) y cada componente contiene el puntero al padre del árbol al que pertenece. Un análisis simple de esta implementación nos permite observar que iniciar+ *n* operaciones añade tiene un coste $\theta(n)$, merge tiene coste $\theta(1)$ y find $O(n)$. No parece que se haya mejorado nada.
Sin embargo, se pueden utilizar dos técnicas que permiten reducir la suma del coste de todas las operaciones que se pueden realizar. La UNIÓN POR RANGO es la primera técnica (por ejemplo, él árbol con menos elementos se cuelga de la raíz del

que tiene más elementos). Al aplicarla se consigue que la altura de los árboles esté acotada por la función log *n* y que, por tanto, **find** cueste θ(log *n*) y **merge** O(1). La segunda de ellas es la COMPRESIÓN DE CAMINOS (aprovechando la operación **find**, que asciende por el árbol hasta encontrar el identificador del conjunto al que pertenece el elemento que estamos buscando, se fuerza a que los elementos de ese camino apunten directamente a la raíz del árbol) que consigue reducir la distancia desde cualquier vértice del árbol a la raíz con lo que, la gran mayoría de las veces, la altura del árbol es menor que *log n*.
La aplicación conjunta de las dos técnicas hace que el coste de efectuar *m* operaciones **find** y **merge**, $n \leq m$, sea $\theta(m \cdot \log^* n)$, donde $\log^* n$ es la función LOGARITMO ITERADO DE N que tiene un crecimiento lentísimo. Esta función se define de la siguiente forma:

$$\log^* 1 = \log^* 2 = 1$$
$$\forall n > 2, \quad \log^* n = 1 + \log^*(\lceil \log_2 n \rceil)$$

Estos son algunos valores de referencia de esta función:
$\log^* 4 = 1 + \log^* 2 = 2$
$\log^* 16 = 1 + \log^* 4 = 3$
$\log^* 60000 = 1 + \log^* 16 = 4$

Como ya se ha mencionado al comienzo de la sección, esta estructura se puede utilizar para resolver el problema de calcular las componentes conexas de un grafo no dirigido. A continuación se muestra el algoritmo correspondiente. El número de **find** y **merge** que efectúa es $m=(n + e)$ y por tanto el coste de este algoritmo es $O((n+e)\cdot \log^* n)$

```
función CCC(g es grafo) dev (m es MFSet)
```
{*Pre:* g es un grafo no dirigido }
```
        m:= iniciar(n);
        Para cada v∈V hacer
                v.visitado:= 'no_visto';
                m:= añade(m, {v});
        fpara
        Para cada v∈V  hacer
                si no_visto(g, v) entonces <g, m>:= CCC-1(g, v, m);
                fsi
        fpara
```
{*Post:* m contiene todas las componentes conexas de g}

```
        dev (m)
ffunción
```

```
función CCC-1(g es grafo; u es vértice; m es MFSet) dev (g es grafo; m es MFSet)
```

{*Pre:* (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g') ∧ (*m* contiene una clasificación de todos los vértices ya vistos en g)}

```
        u.visitado:= 'visto';
        v1:= find(m, u);
        Para cada w ∈ ady(g,u) hacer
                v2:= find( m, w);
                si visto(g, w) entonces
                        si (v1 ≠ v2) entonces m:= merge(m, v1, v2);
                        fsi
                sino    m:= merge(m, v1, v2);
                         <g, m>:= CCC-1(g, w, m);
        fpara
```

{*Post:* Visto(g,u) ∧ marcados a visto todos los vértices de g accesibles desde u por caminos formados exclusivamente por vértices que no estaban vistos en g', según el orden de visita del recorrido en profundidad. Además todos los vértices ya vistos están convenientemente clasificados en m}

```
        dev ( g, m )
ffunción
```

3.2. **Recorrido en anchura**

El algoritmo de recorrido en anchura, en inglés *breadth-first search* y que a partir de ahora escribiremos BFS para abreviar, se caracteriza porque permite recorrer completamente el grafo. El BFS visita los vértices y las aristas las mismas veces que lo hace el DFS pero en un orden distinto. En el BFS el orden de recorrido es tal que los vértices se visitan según el orden primero en anchura o por niveles. Este orden implica que, fijado un vértice, primero se visitan los vértices que se encuentran a distancia mínima uno de él, luego los que se encuentran a distancia mínima dos, etc.

Para poner en marcha el recorrido se elige un vértice cualquiera como punto de partida. No importa si el grafo es dirigido o no, en cualquier caso el algoritmo acaba cuando todos los vértices han sido visitados.

Una vez visto el recorrido en profundidad es sencillo obtener el algoritmo que efectúa el recorrido en anchura de un grafo. Basta con obtener una versión iterativa del DFS y sustituir la pila por una cola.

```
acción REC_PROF_ITERATIVO(g es grafo; u es vértice)
{Pre: g es un grafo no dirigido y no_visto(g,u)}
     p:= pila_vacia;   p:= apilar(p,u);
     mientras no_vacia(p) hacer
           u:= cima(p); p:=desapilar (p);
           si no_visto(g,u) entonces
                u.visitado:= 'visto';
                para cada w ∈ ady(g,u) hacer
                     si no_visto(g,w) entonces
                          p:= apilar(p, w);
                     fsi
                fpara
           fsi
     fmientras
{Post: Todos los vértices de la misma componente conexa a la que pertenece u han sido marcados a 'visto' en el orden que sigue el recorrido en profundidad de g}
facción
```

Versión iterativa del DFS

Una primera versión del BFS, traducción directa del DFS iterativo, se presenta a continuación.

```
acción REC_ANCHO(g es grafo)
```

{*Pre:* g es un grafo no dirigido}

```
        Para cada v∈V hacer  v.visitado:='no_visto' fpara
        Para cada v∈V hacer
                si no_visto(g,v) entonces g:= REC_ANCHO_1(g,v)
                fsi
        fpara
```

{*Post:* Todos los vértices de g han sido marcados a 'visto' en el orden que sigue el recorrido en anchura de g}

```
faccción
```

```
función REC_ANCHO_1(g es grafo; u es vértice) dev (g es grafo)
```

{*Pre:* (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g')}

```
    c := cola_vacia;
    c := pedir_turno(c,u);
    mientras no_vacia(c) hacer
        u := primero(c);
        c := avanzar(c);
        si no_visto(g,u) entonces
            u.visitado:= 'visto';
            para cada w ∈ ady(g,u) hacer
                si no_visto(g,w) entonces
                    c:= pedir_turno(c, w);
                fsi
            fpara
        fsi
    fmientras
```

{*Post:* Visto(g,u) ∧ marcados a 'visto' todos los vértices de g accesibles desde u que no estaban vistos en g según el orden de visita del recorrido en anchura }

```
    dev ( g )
ffunción
```

Notar que en la cola del BFS se guardan los vértices adyacentes 'no_vistos' del último vértice que ha pasado a 'visto' y que es posible que existan vértices repetidos en la cola. También puede pasar que al extraer un vértice de la cola éste ya esté marcado a 'visto' y que no haga falta tratarlo (ni a sus adyacentes tampoco). El coste del algoritmo para un grafo implementado con listas de

adyacencia es $\theta(n+e)$, el mismo que el DFS, aunque requiere más espacio debido a la cola de vértices que utiliza, que en el caso peor puede ocupar $\theta(e)$.

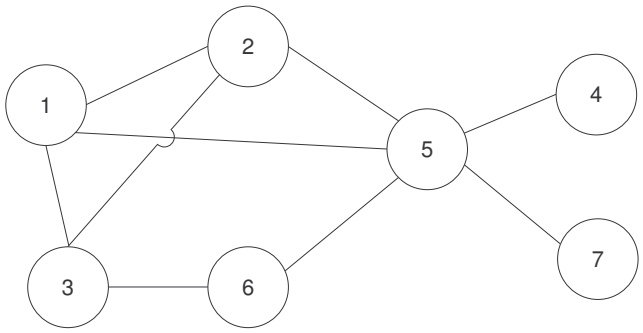
La siguiente versión del BFS garantiza que no hay vértices repetidos en la cola porque se marcan a `'visto'` antes de entrar en ella. En realidad la marca de `'visto'` se sustituye por la de `'en_cola'` para poder preguntar si está en la cola y que no se produzcan repeticiones. Además, cuando un vértice sale de la cola es porque todos sus hijos sin visitar ya han entrado. Aprovechamos el recorrido para calcular a qué distancia (distancia mínima) se encuentra cualquier vértice del grafo del vértice por el que se ha comenzado el recorrido (`u.distancia`).

```
función REC_ANCHO_1_OTRO(g es grafo; u es vértice) dev (g es grafo)
{Pre: (u∈ V) ∧ ¬Visto(g,u) ∧ (g=g')}
     c := cola_vacia;  u.distancia:= 0;
     c := pedir_turno(c,u); u.en_cola:= CIERTO;
     mientras no_vacia(c) hacer
          u := primero(c); c :=avanzar(c); u.visitado:= 'visto';
          /* en este punto le podríamos asociar al vértice u la numeración en el orden del recorrido en
          anchura que es cuando sale de la cola */
          para cada w ∈ ady(g,u) hacer
               si no(w.en_cola) entonces
                    w.en_cola:= CIERTO;
                    w.distancia:= u.distancia + 1;
                    /* la distancia del hijo a la raíz del recorrido, vértice u, es la de su padre +1 */
                    c:= pedir_turno(c, w);
               fsi
          fpara
     fmientras
{Post: Visto(g,u) ∧ marcados a 'visto' todos los vértices de g accesibles desde u que no estaban vistos en g
según el orden de visita del recorrido en anchura }
     dev ( g )
ffunción
```

El orden en que se visitan los vértices en esta versión viene dado por el instante en que salen de la cola. Evitando repeticiones en la cola, el espacio necesario se reduce en el caso peor a $\theta(n)$. En la cola a lo sumo habrá vértices pertenecientes a

dos niveles consecutivos del grafo, los que se encuentran a distancia mínima *k* y a distancia mínima *k+*1 del vértice a partir del cual se ha iniciado el recorrido.

Como en el DFS, en el BFS la secuencia de vértices producida por el recorrido en anchura no es única sino que depende de cuál es el vértice elegido para comenzar el recorrido y de cómo se van eligiendo los adyacentes.
La siguiente figura muestra algunas de las secuencias que puede producir el BFS sobre un grafo no dirigido. Para obtener la primera secuencia se ha comenzado por el vértice 1, para la segunda por el 5 y para la tercera por el 7. Siempre se han generado los adyacentes en el orden que marca su etiqueta.



Recorrido 1: {1, 2, 3, 5, 6, 4, 7}
Recorrido 2: {5, 1, 2, 4, 6, 7, 3}
Recorrido 3: {7, 5, 1, 2, 4, 6, 3}

También podemos numerar los vértices en el orden del recorrido en anchura y podemos construir el árbol asociado al recorrido en anchura, T_{BFS}, que se construye igual que el T_{DFS}.

Algunas características destacables del T_{BFS} se citan a continuación. Sea G=<V,E> un grafo y sea T_{BFS}=<V,F> el árbol asociado al recorrido en anchura de G.
Definición: se define la distancia más corta entre los vértices *s* y *v*, $\delta(s,v)$, como el mínimo número de aristas en cualquier camino entre *s* y *v* en G.
Teorema: Para todo vértice $w \in V$, la longitud del camino desde la raíz hasta *w* en el T_{BFS} coincide con la longitud del camino más corto desde la raíz hasta *w* en G ➔ `w.distancia` = $\delta(s,w)$ siendo *s* la raíz del T_{BFS}.

Un par de lemas necesarios para la demostración del teorema (que no haremos).
Lema 1: Sea $s \in V$ un vértice arbitrario de G, entonces para cualquier arista $(u,v) \in E$ sucede que $\delta(s,v) \leq \delta(s,u) + 1$.

Prueba:

- Si no existe camino desde *s* hasta *v*, tampoco existe camino desde *s* hasta *u* porque, en caso contrario, desde *s* se llegaría a *u* y desde *u* a *v*. Entonces el lema es trivialmente cierto.
- Si existe camino desde *s* hasta *v*, no puede ser que $\delta(s,v) > \delta(s,u)$ porque existe un camino más corto que llega a *v* pasando por *u* y que mide exactamente $\delta(s,u)+1$.
- Si $\delta(s,v) < \delta(s,u)$ entonces ya se satisface directamente el lema.

Lema 2: Si durante la ejecución del BFS la cola Q contiene los vértices $<v_1, v_2, \ldots, v_k>$, donde v_1 es el primero de Q y v_k es el último, entonces v_k.distancia ≤ v_1.distancia+1 y v_i.distancia ≤ v_{i+1}.distancia para i=1, 2, …, k-1.

Prueba: Haciendo inducción sobre el número de operaciones sobre la cola.

- Una operación: cuando la cola contiene sólo a la raíz el lema es trivialmente cierto.
- Supongamos que el lema es cierto para *j*-1 operaciones.
- Paso inductivo: Efectuamos la operación *j*-ésima eliminando el vértice v_1 de la cola. Ahora el nuevo primer vértice es v_2. Por hipótesis de inducción se tiene que v_k.distancia ≤ v_1.distancia+1 ≤ v_2.distancia+1.
  Por tanto el nuevo primer vértice de la cola cumple v_k.distancia ≤ v_2.distancia+1.
  Si la operación *j*-ésima es insertar el vértice *w* en la cola, *w* pasa a ser el v_{k+1} y su padre es v_1, el que ha salido de la cola, y tenemos que v_{k+1}.distancia = v_1.distancia+1 por lo que el lema sigue siendo cierto cuando se efectúa la operación *j*-ésima sobre la cola (insertar *w*).

Hay que mencionar que el árbol asociado al recorrido en anchura, T_{BFS}, también permite clasificar las aristas del grafo de partida. Si se trata de un grafo no dirigido entonces existen aristas de dos tipos: *tree edges* y *cross edges*; estas últimas son las que permiten detectar la existencia de ciclos. Sin embargo, si se trata de un grafo dirigido, se tienen aristas de tres tipos: *tree edges*, *back edges* y *cross edges* con la particularidad de que éstas últimas pueden ir tanto de derecha a izquierda como de izquierda a derecha. No existen *forward edges*.

Una de las aplicaciones clásicas del recorrido en anchura es la de, dado un grafo, encontrar el camino más corto (con menos aristas) entre dos vértices dados. Esta

aplicación es consecuencia inmediata de los lemas enunciados anteriormente que garantizan que el BFS llega a un vértice siempre por el camino más corto posible desde el vértice del que parte el recorrido.

3.2.1. Algunas aplicaciones particulares

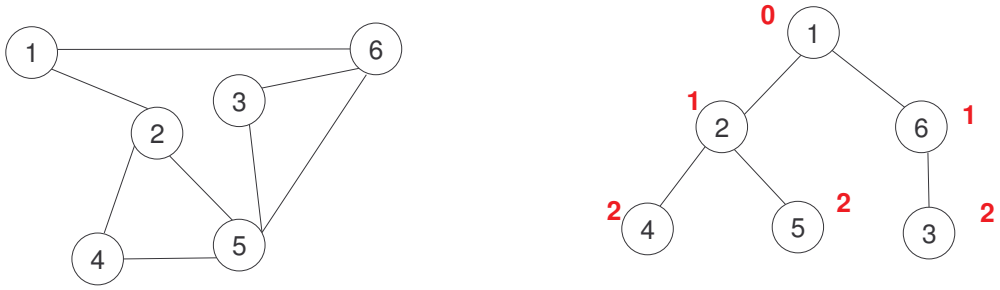
El BFS se puede utilizar para 'medir' el grafo. Las medidas que vamos a definir son: excentricidad, diámetro, radio y centro.

Sea G=<V,E> un grafo no dirigido:

Definición: la excentricidad de un vértice *v*, $v \in V$, se define como la distancia máxima desde *v* a cualquier otro vértice del grafo G siguiendo caminos de longitud mínima.

$$\text{excentricidad}(v) = \text{MAX}\ \{\forall w : w \in V \wedge v \neq w : \delta(v, w)\}$$

Si lanzamos un BFS desde *v*, calcularemos la distancia mínima entre *v* y todos los demás vértices del grafo. Basta con devolver el máximo de todas esas distancias y ya tendremos la excentricidad de *v*.



Grafo G y un posible TBFS de G. En el árbol se ha anotado fuera de los nodos la distancia mínima de cada uno de los vértices a la raíz. La excentricidad del vértice 1 es 2.

Definición: El diámetro de G es el máximo de las excentricidades de todos sus vértices.

$$\text{diámetro}(G) = \text{MAX}\ \{\forall w : w \in V : \text{excentricidad}(w)\}$$

Definición: El radio de G es el mínimo de las excentricidades de todos sus vértices.

$$\text{radio}(G) = \text{MIN}\ \{\forall w : w \in V : \text{excentricidad}(w)\}$$

Definición: El centro de G está formado por todos aquellos vértices cuya excentricidad coincide con el radio de G.

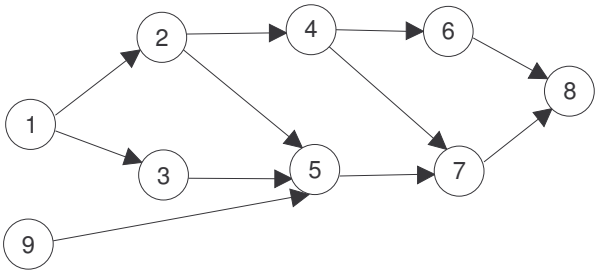
$$\text{centro}(G) = \{\ w \in V : \text{excentricidad}(w) = \text{radio}(G)\}$$

Un grafo completo tiene radio(G)=diámetro(G)=1.

3.3. **Ordenación topológica**

Una ordenación topológica de los vértices de un grafo DIRIGIDO y ACICLICO es una secuencia en la que aparecen todos los vértices del grafo y que cumple que si $(u,v) \in E$ entonces *u* aparece antes que *v* en la secuencia. El algoritmo de ordenación topológica, *Topological Sort*, genera una ordenación lineal de todos los vértices del grafo que satisface las condiciones de la ordenación. Este algoritmo funciona sólo sobre grafos DIRIGIDOS y ACICLICOS, DAG.

Dado un grafo G=<V,E> DAG podemos generar diferentes ordenaciones lineales que sean ordenaciones topológicas. Todo depende del vértice de partida y de cuál se elija como siguiente vértice a visitar. Para el grafo que se presenta en la siguiente figura, resulta que una ordenación posible es O_1 = {1, 2, 4, 3, 9, 5, 7, 6, 8}, mientras que otra es O_2 = {9, 1, 3, 2, 5, 4, 6, 7, 8} aunque existen más ordenaciones posibles.



Podemos aplicar dos aproximaciones distintas para diseñar un algoritmo que calcule una ordenación topológica. La primera de ellas se basa en el hecho de que un vértice, $v \in V$, sólo puede aparecer en la secuencia cuando todos sus predecesores han sido visitados (ya aparecen en la ordenación) y, a partir de ese momento, *v* puede aparecer en cualquier lugar posterior. Una aplicación inmediata de esta idea conduce a llevar un contador para cada vértice de modo que indique cuántos de sus predecesores faltan por aparecer en la secuencia; en el momento en que ese contador sea cero, el vértice en cuestión ya puede aparecer.
La segunda aproximación es justo la lectura inversa de la primera: Un vértice, $v \in V$, ocupa un lugar definitivo en la secuencia cuando toda su descendencia ha sido visitada y siempre irá delante de todos ellos, de hecho, esa es la posición más alejada del comienzo de la secuencia en la que puede aparecer, más lejos ya no es posible.

El siguiente algoritmo es el que se obtiene de la primera aproximación, el de la segunda versión es una aplicación inmediata del recorrido en profundidad.

```
función ORD-TOPO(g es grafo) dev (s es secuencia(vértices))
```
{*Pre:* g es un grafo dirigido y sin ciclos }
```
        c:= conj_vacio;
        Para cada v∈V  hacer
                NP[v]:= 0;
        fpara
        Para cada v∈V  hacer
              Para cada w ∈ suc(g,v)  hacer
                 NP[w]:= NP[w] + 1;
              fpara
        fpara
        Para cada v∈V  hacer
                si (NP[v]=0) entonces c:= insertar(c, w);
                fsi
        fpara
```
/* para cada v∈V, NP[v] contiene el número predecesores y c contiene sólo aquellos vértices que tienen cero predecesores. El recorrido puede comenzar por cualquiera de ellos */
```
        s:= sec_vacia;
        mientras ¬vacio(c) hacer
                v:= obtener_elem(c);
                c:= eliminar(c, v);
                s:= concatenar(s, v);
                Para cada w ∈ suc(g,v)  hacer
                      NP[w]:= NP[w]-1;
                      si NP[w]=0 entonces c:= añadir(c, w);
                      fsi
                fpara
        fmientras
```
{*Post: s* = orden_topológico(g) }
```
        dev (s)
ffunción
```

El coste de este algoritmo es el mismo que el del recorrido en profundidad sobre listas de adyacencia, $\theta(n+e)$, si las operaciones sobre el conjunto tienen coste $\theta(1)$; basta con guardar el conjunto en una pila y ya se consigue ese coste.