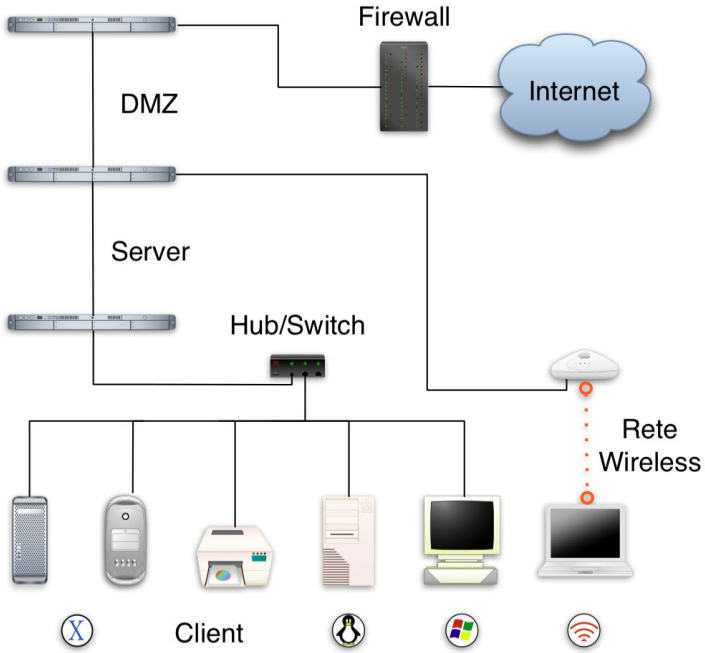
# Tema 6
# Grafos I

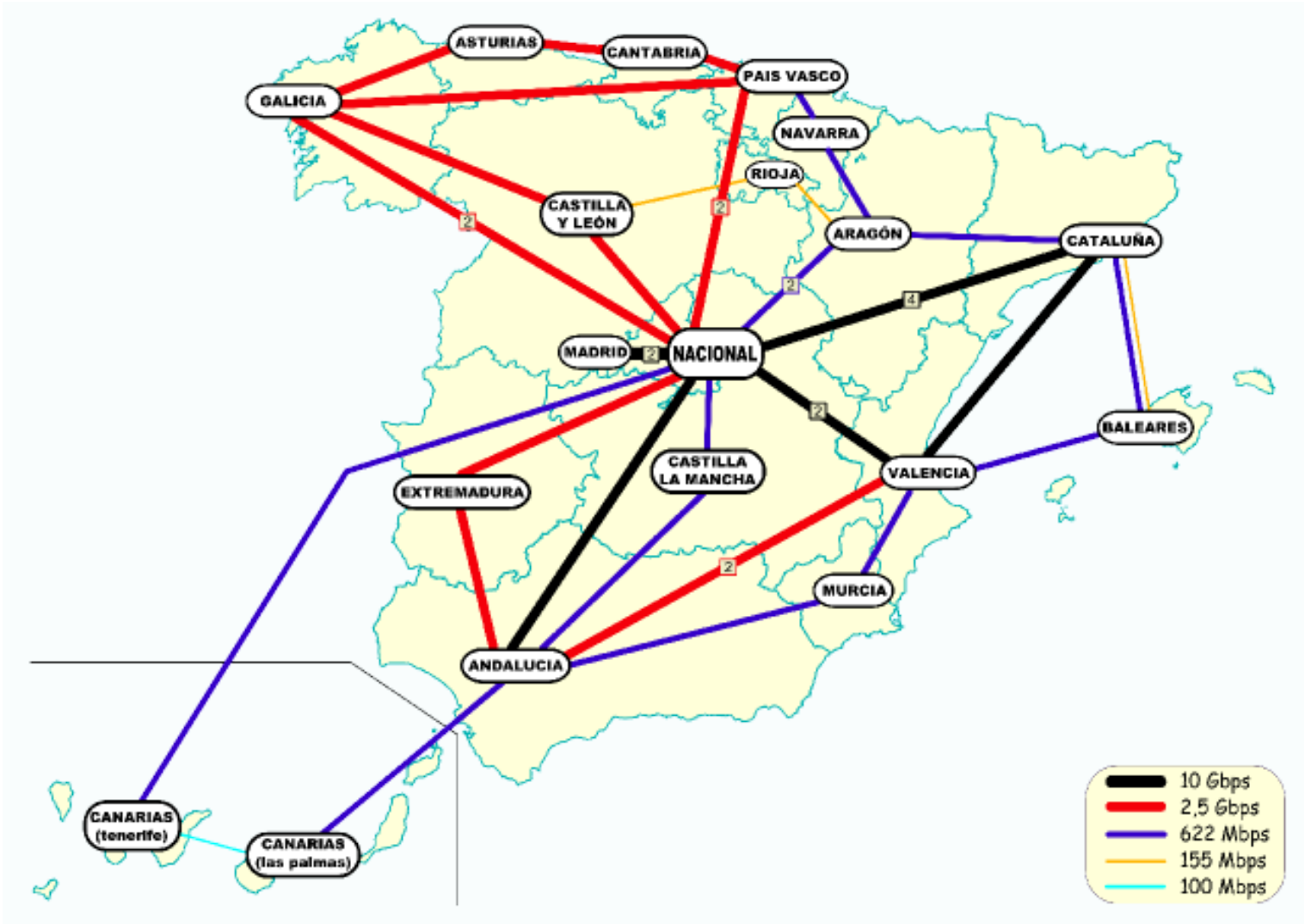
1. Conceptos y definiciones
2. Representación de grafos
3. TAD grafo
4. Recorrido de un grafo
5. Componentes conexas y fuertemente conexas
6. Matriz de caminos
7. Puntos de articulación

## Red local

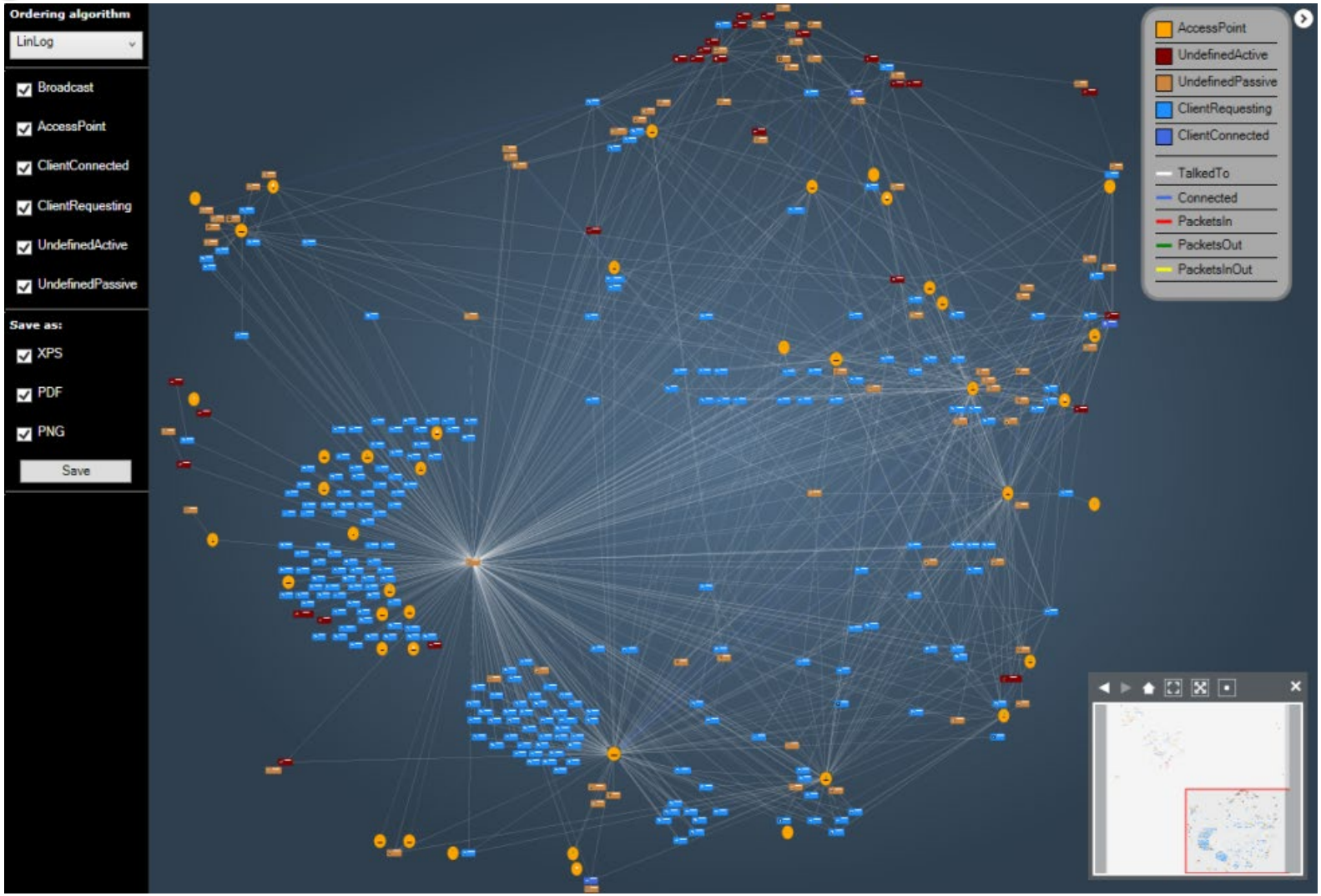



## Grafo de precedencia para la construcción de una casa

# RedIRIS antigua: Grafo valorado

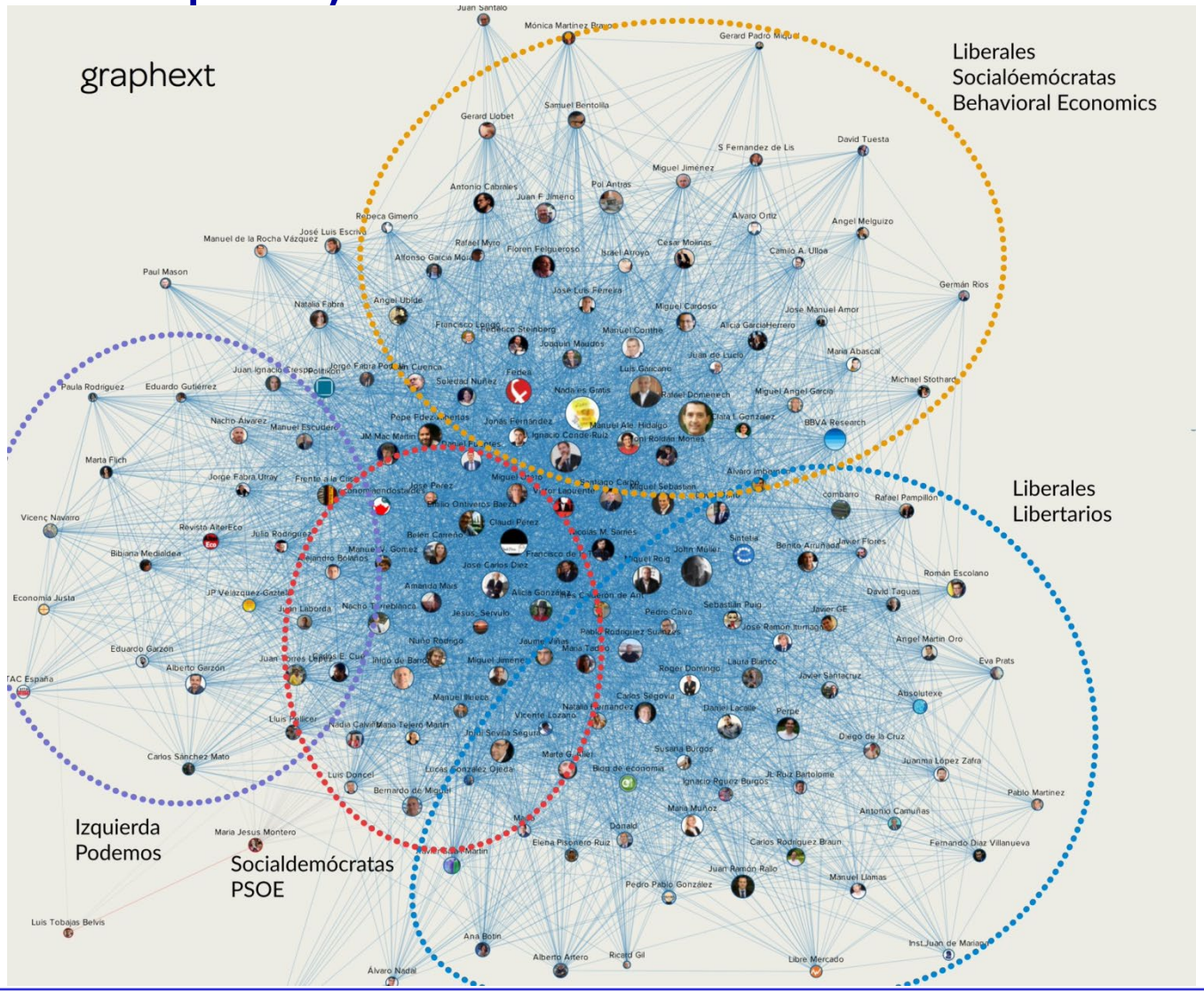


# Tarlogic: Grafo de actividad de dispositivos WiFi



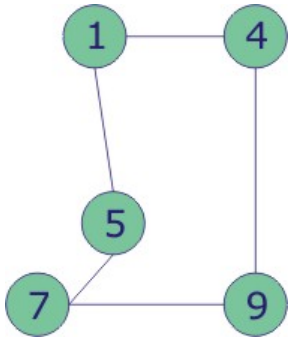

https://www.tarlogic.com/productos/wireless-audit-framework/grafo-de-actividad-de-dispositivos-wifi

## Cuentas más relevantes sobre Economía en España con @graphext (tweet de Victoriano Izquierdo)

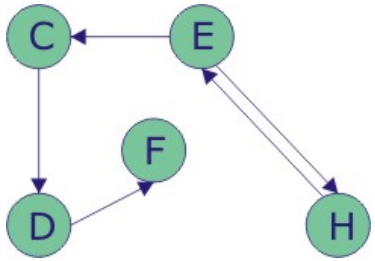


# 2. Conceptos y definiciones

- **Grafo G=(V,A)**
  - Conjunto de vértices o nodos **V** y un conjunto de arcos **A.**
- **Arco/arista (u,v):** formado por un par de nodos u,v
  - Grafo dirigido (digrafo): Arcos u→v
  - Grafo no dirigido: Arcos u-v
- **Nodos adyacentes** (se relacionan por un arco)
  - Grafo dirigido: u→v (u es adyacente a v, v es adyacente de u)
  - Grafo no dirigido: u-v (u y v son adyacentes)



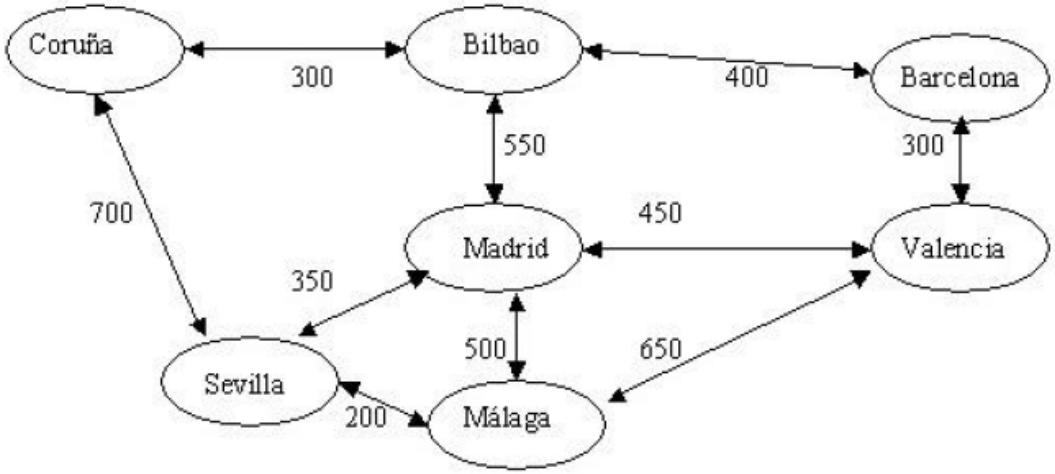

**Grafo no dirigido G=(V,A)**
V={1,4,5,7,9}
A={1-4, 5-1, 7-9, 7-5, 4-9}

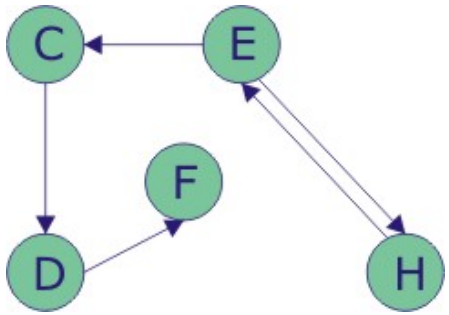



**Grafo dirigido G=(V,A)**
V={C,D,E,F,H}
A={C→D, D→F, E→H, H→E, E→C}

- **Grafo valorado**
  - Sus arcos tienen un factor de peso asociado.
- **Grado** de un nodo v:
  - Grafo no dirigido: Grado(v)=nº aristas que contiene a v
  - Grafo dirigido
    - Grado de entrada: gradent(v)=nº de arcos que llegan a v
    - Grado de salida: gradsal(v)=nº de arcos que salen de v

Grado(Madrid)=4

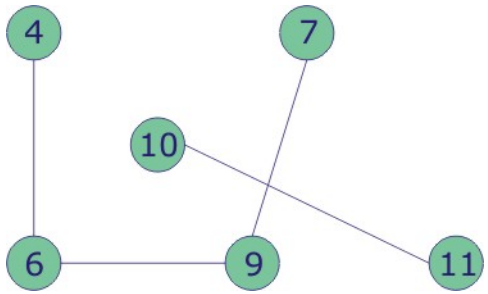


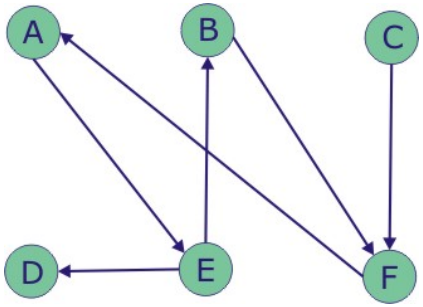

gradent(D)=1
gradsal(D)=1

# 2. Conceptos y definiciones

- **Camino**
  - Secuencia de *n+1* vértices: P=($v_0$, $v_1$, $v_2$, … , $v_n$) tal que ($v_i$,$v_{i+1}$)∈*A*(arcos).
  - Longitud del camino: Nº de arcos que lo forman.
- **Bucle**
  - Camino v-v: cuando existe arco de un vértice a sí mismo (v,v)
- **Camino simple**
  - Si todos los nodos que forman el camino son distintos, pudiendo ser iguales $v_0$, $v_n$ (los extremos del camino).
- **Ciclo**
  - Camino simple cerrado, $v_0$=$v_n$ de longitud ≥ 2 (⇒al menos 3 nodos)
  - Ciclo de longitud k: k-ciclo




$P_1$=(4,6,9,7) es un camino de longitud 3




(A→E→B→F→A) forman un ciclo de longitud 4 (4-ciclo)

- ¿Tipos de datos para representar los grafos en memoria?
  - Hay que tener en cuenta que se debe representar un número (finito) de vértices y de arcos que unen dos vértices.
  - Representación secuencial (arrays): MATRIZ DE ADYACENCIA
  - Representación dinámica (estructura multienlazada): LISTA DE ADYACENCIA
  - La elección de una representación u otra dependerá de las operaciones que se apliquen sobre los vértices y arcos.

## MATRIZ DE ADYACENCIA

- Sea G=(V,A) un grafo de *n* nodos.
- Suponemos que los nodos $V=\{v_1, v_2, \ldots, v_n\}$ están ordenados y podemos representarlos por sus ordinales $\{1, 2, \ldots, n\}$.
- La representación de los arcos se hace con una matriz **A** de $n \times n$ elementos $a_{ij}$ definida como:

$$a_{ij} = \begin{cases} 1 & si\ hay\ un\ arco\ (v_i, v_j) \\ 0 & si\ no\ hay\ arco\ (v_i, v_j) \end{cases}$$

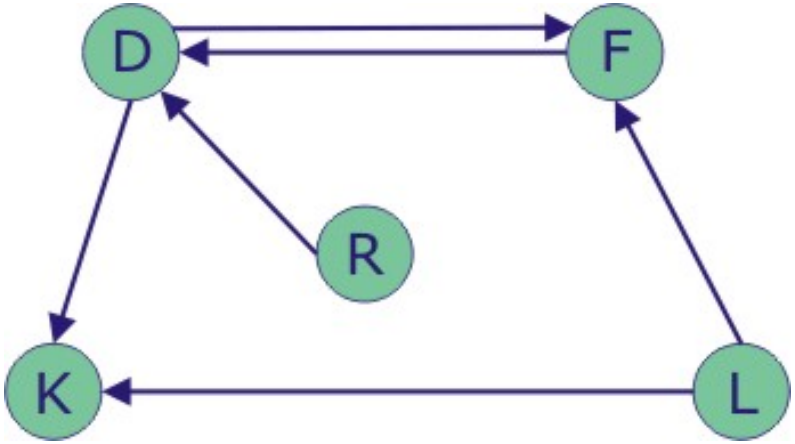
**A** se denomina ***matriz de adyacencia***.

- En ocasiones esta matriz es booleana con elementos $a_{ij}$ tomando valores CIERTO si existe el arco $v_i \rightarrow v_j$ y FALSO en caso contrario.

## MATRIZ DE ADYACENCIA

- **Ejemplo 1: Grafo dirigido**

  Si el orden de los vértices es {D, F, K, L, R}, la matriz de adyacencia es la siguiente:



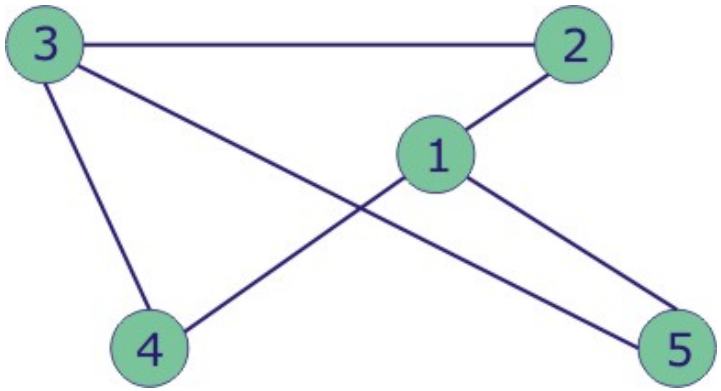

$$A = \begin{array}{c} \begin{matrix} D & F & K & L & R \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} D \\ F \\ K \\ L \\ R \end{matrix}$$

**R→D**

## MATRIZ DE ADYACENCIA

- **Ejemplo 2: Grafo no dirigido.**

  La matriz de adyacencia es simétrica ya que cada arco no dirigido $v_i$-$v_j$ se corresponde con los arcos dirigidos $v_i \rightarrow v_j$, $v_j \rightarrow v_i$



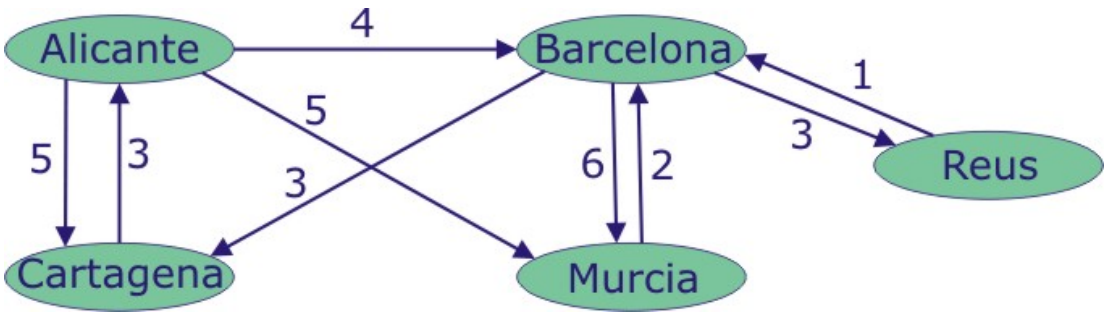

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

## MATRIZ DE ADYACENCIA

- **Ejemplo 3: Grafo valorado:**

  Si existe arco, $a_{ij}$ es el factor de peso; la no existencia de arco supone que $a_{ij}$ es **0** ó ∞ (esto sólo si el factor de peso puede ser 0). A esta matriz se la denomina *matriz valorada o matriz de pesos*.

  Si suponemos los vértices en el orden *V={Alicante, Barcelona, Cartagena, Murcia, Reus}*, la matriz de pesos *P* será:




$$P = \begin{pmatrix} 0 & 4 & 5 & 5 & 0 \\ 0 & 0 & 3 & 6 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

## MATRIZ DE ADYACENCIA: Representación en C

- Representación estática → debemos prever el nº máximo de nodos (MAXVERTICES).
- Los nodos o vértices tienen un identificador que se guarda en un vector de cadenas (VERTICES).
- La matriz de adyacencia A que se representa es de tipo int, tomando los valores 1 ó 0, según haya arco o no entre un par de vértices.
- El entero N indica cuántos vértices hay en el grafo.
- Si el grafo tiene factor de peso numérico, puede asociarse directamente a la matriz.

```
struct tipografo{
    int N;
    char VERTICES[MAXVERTICES][LONGITUDCADENA];
    int A[MAXVERTICES][MAXVERTICES];
};

typedef struct tipografo * grafo;
```

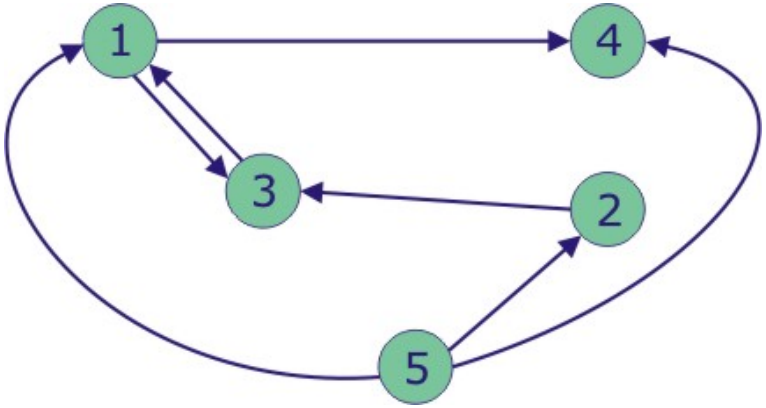
## MATRIZ DE ADYACENCIA

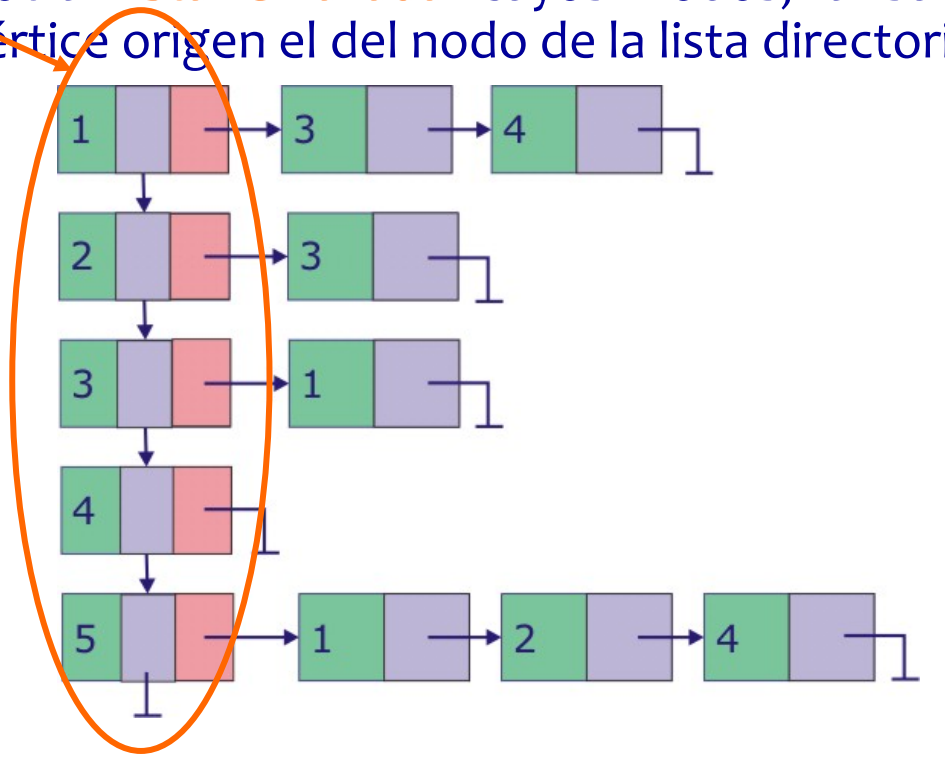
- **Ventajas**
  - El orden de eficiencia de las operaciones de obtención del coste asociado a un arco.
  - La comprobación de conexión entre 2 vértices cualesquiera es independiente del número de vértices y arcos del grafo

- **Desventajas**
  - Es una representación que no modifica el número de sus vértices ya que una matriz no permite que se le supriman filas o columnas.
  - Grafo disperso, con pocos arcos → muchos ceros en la matriz de adyacencia (matriz *sparse*)
    - Mucho espacio desaprovechado, pues la matriz siempre ocupa lo mismo, haya o no muchos arcos entre nodos.

## LISTAS DE ADYACENCIA

- Representación de los grafos que soluciona las ineficacias de la matriz de adyacencia cuando existen pocos arcos
- Estructura multienlazada formada por:
  - Una **lista directorio** cuyos nodos representan los vértices del grafo
  - De estos nodos emerge otra **lista enlazada** cuyos nodos, a su vez, representan los arcos con vértice origen el del nodo de la lista directorio.



## ¿MATRICES DE ADYACENCIA O LISTAS DE ADYACENCIA?

- Uso de memoria
  - N: número de vértices, a: número de arcos
  - Con listas de adyacencia, memoria usada: O(N+a)
  - Con matrices de adyacencia, memoria usada: $O(N^2)$
  - ¿Cuál usa menos memoria?
- Ventajas listas adyacencia
  - **Más adecuada cuando $a<<N^2$.** *¿Cuánto es mucho menor?*
- Inconvenientes listas adyacencia
  - Representación más compleja.
  - Es ineficiente para encontrar las aristas que llegan a un nodo.

# 4. TAD Grafo

- Operaciones:
  - **crear_arco(X,Y)** añade el arco (X,Y) al grafo
  - **borrar_arco(X,Y)** elimina del grafo el arco (X,Y)
  - **es_adyacente(X,Y)** es una función lógica que devuelve TRUE si los vértices (X,Y) forman un arco
  - **crear_vertice(X)** añade el vértice X al grafo G
  - **borrar_vertice(X)** elimina el vértice X del grafo G
- Estas operaciones son el núcleo a partir del cual se construye el grafo.

- Realización con matriz de adyacencia

```
void crear_arco(grafo *G, int V1, int V2){
  G->A[V1][V2]=1;
}
```

Arco v1→v2=> Elemento (v1,v2) de la matriz de adyacencia = 1

```
void borrar_arco(grafo *G, int V1, int V2)
{
  G->A[V1][V2]=0;
}
```

Arco v1→v2=> Elemento (v1,v2) de la matriz de adyacencia = 0

```
int es_adyacente(grafo G, int V1, int V2)
{
   return (G->A[V1][V2]);
}
```

Devuelve el valor de Arco v1→v2

- Realización con matriz de adyacencia

```
void crear_vertice(grafo *G, char *Vert)
{
     int i;
     if (G->N <= MAXVERTICES){
         (G->N)++;
         strcpy(G->VERTICES[(G->N)-1],Vert);
         for (i=0;i<G->N;i++){
           G->A[i][(G->N)-1]=0;
           G->A[(G->N)-1][i]=0;
         }
     }
     else
         printf("Grafo lleno!\n");
}
```

Incremento N (nº de vértices)

Añado el nuevo vértice Vert a la lista de vértices G->VERTICES

El nuevo vértice NO tiene arcos => valores en matriz de adyacencia = FALSE

## Realización con matriz de adyacencia

Esta operación no sólo supone eliminar el vértice de la lista de vértices, sino que además se debe ajustar la matriz de adyacencia, ya que el índice de los vértices posteriores al eliminado debe quedar *decrementado en 1*

`posicion(G,Vert)` devuelve la posición de un vértice `Vert` en la lista V de vértices de G

Muevo elementos de G->VERTICES a la izquierda para cubrir el hueco dejado por Vert

Muevo columnas de G->A a la izquierda para cubrir el hueco dejado por Vert

Muevo filas de G->A hacia arriba para cubrir el hueco dejado por Vert

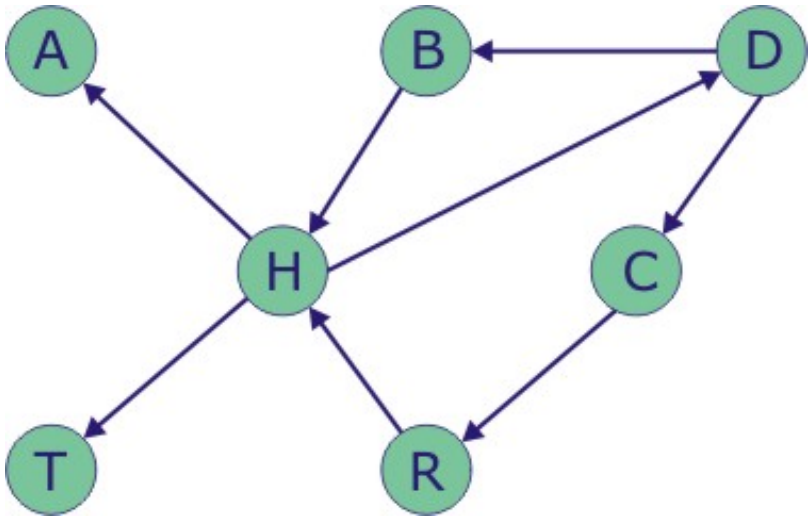
Decremento N

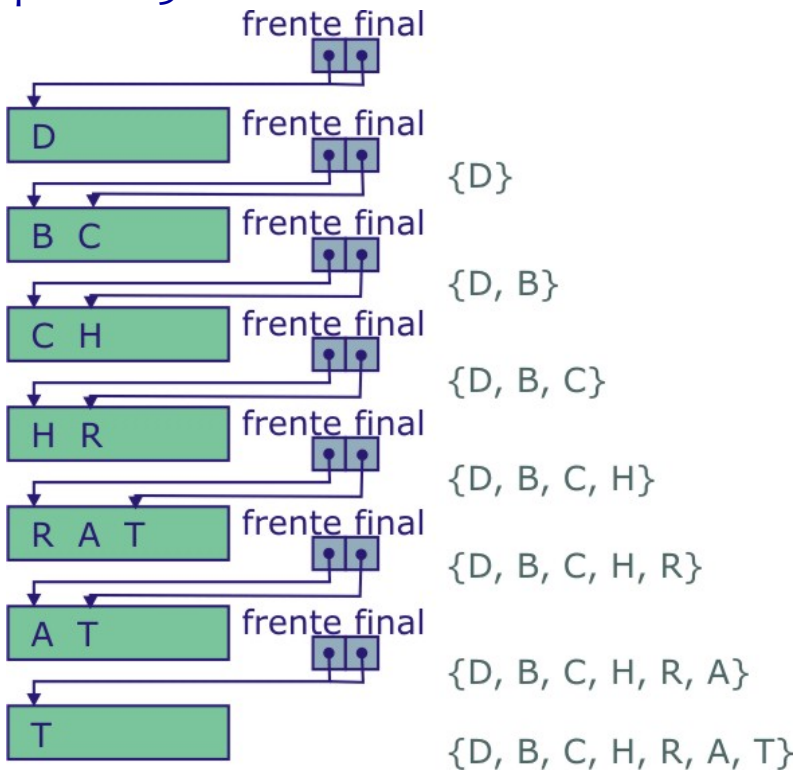
```
void borrar_vertice(grafo *G, char *Vert)
{
      int F,C,P;
      P=posicion(G,Vert);
      if (P>=0 && P< G->N){
          for (F=P;F<(G->N)-1;F++)
            G->VERTICES[F]=G->VERTICES[F+1];

          for (C=P;C<(G->N)-1;C++)
            for (F=0; F<G->N; F++)
             G->A[F][C]=G->A[F][C+1];

           for (F=P;F<(G->N)-1;F++)
            for (C=0; C<G->N; C++)
             G->A[F][C]=G->A[F+1][C];

          (G->N)--;
      }
      else
          printf("Vertice inexistente!\n");
}
```

- **Recorrido:** consiste en visitar cada uno de los nodos a partir de uno dado.
  - Recorrido en anchura (no recursivo)
    - Comienza visitando el vértice de partida V, para a continuación visitar los adyacentes que no estuvieran ya visitados. Así sucesivamente con los adyacentes.
    - Utiliza una **cola** como estructura auxiliar en la que se mantienen los vértices que se vayan a procesar posteriormente.
  - Recorrido en profundidad recursivo
    - Comienza visitando el vértice de partida V y después se recorre en profundidad cada vértice adyacente a V no visitado; así hasta que no haya más vértices adyacentes no visitados.
  - Recorrido en profundidad no recursivo
    - Definición no recursiva ⇒ **pila** como estructura auxiliar para guardar vértices adyacentes no visitados.

# 5. Recorrido de un grafo

- **Recorrido en anchura: usa una cola como estructura auxiliar**
  1. Visitar el vértice de partida.
  2. Meter en la cola el vértice de partida y marcarlo como procesado.
  3. MIENTRAS QUE LA COLA NO ESTÉ VACÍA
     1. Retirar el nodo frente de la cola, y visitarlo.
     2. Meter en la cola todos los vértices adyacentes al nodo retirado que no estén procesados y marcarlos como procesados.
  4. Si queda algún vértice no visitado, repetir los pasos 2-3



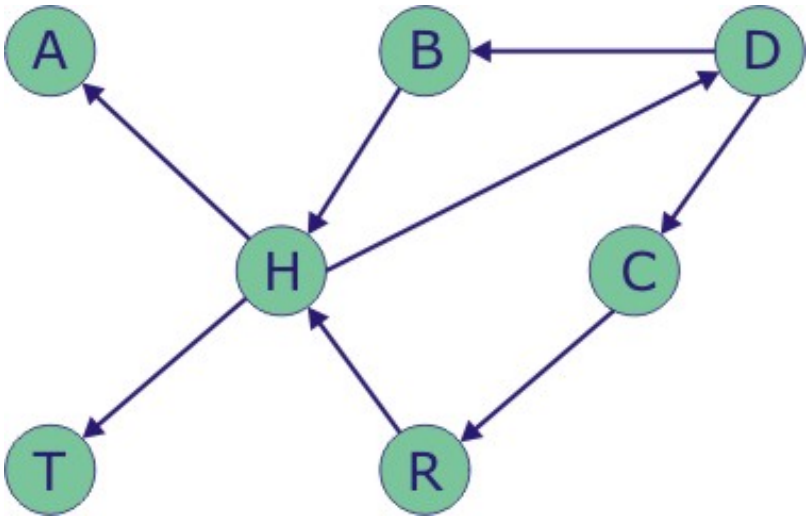



# 5. Recorrido de un grafo

- **Recorrido en profundidad RECURSIVO**

```
Para todos los vértices vi
    Si vi sin visitar
        RecorridoRecursivo(vi)
    Fin
Fin
```

```
RecorridoRecursivo(u)
    Marcar u como visitado
    Para cada vértice vi adyacente a u
        SI vi no visitado
            RecorridoRecursivo(vi)
        Fin
    Fin
```



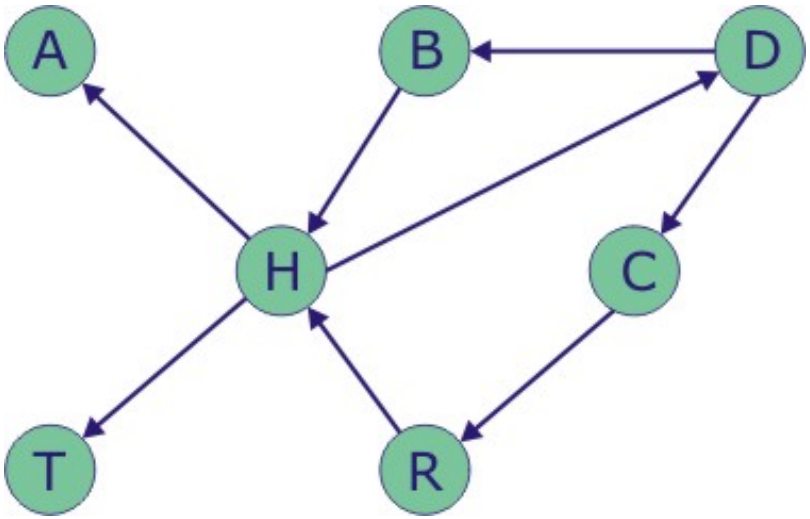

D →RR(D) {D} Adyacentes: {B,C}
→RR(B) {D,B} Adyacentes: {H}
→RR(H) {D,B,H} Adyacentes: {A,T}
→RR(A) {D,B,H,A} Adyacentes: {}
→RR(T) {D,B,H,A,T} Adyacentes:{}
→RR(C) {D,B,H,A,T,C} Adyacentes:{R}
→RR(R) {D,B,H,A,T,C,R} Adyacentes: {}

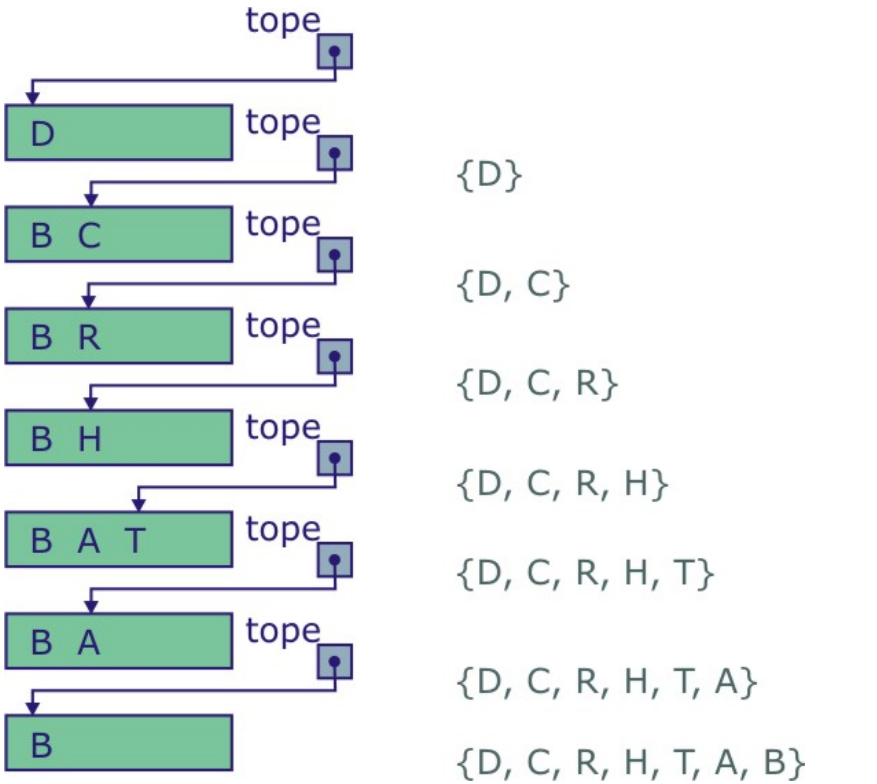
No hay nodos sin visitar, por lo que termina el recorrido:

{D,B,H,A,T,C,R}

# 5. Recorrido de un grafo

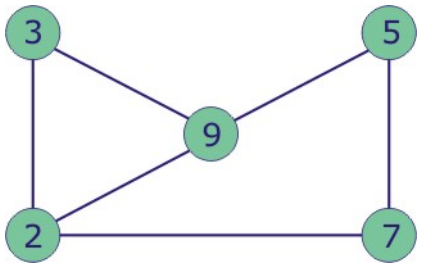
- **Recorrido en profundidad NO RECURSIVO: usa una pila como elemento auxiliar**
  1. Visitar el vértice de partida.
  2. Meter en la pila el vértice de partida y marcarlo como procesado.
  3. Mientras que la pila no esté vacía
     1. Retirar el nodo tope de la pila y procesarlo.
     2. Meter en la pila todos los vértices adyacentes al nodo leído del tope que no estén procesados y marcarlos como procesados.
  4. Si queda algún vértice no visitado, repetir a partir de él los pasos 2-3.



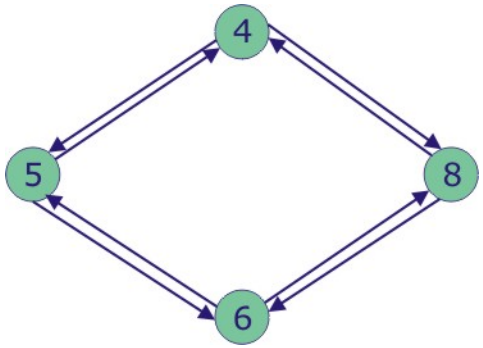



# 6. Componentes conexas de un grafo

- **Grafo no dirigido**
  - **Conexo** si existe un camino entre cualquier par de nodos que forman el grafo.
  - Si el grafo no es conexo, se pueden determinar las **componentes conexas**
- **Grafo dirigido**
  - **Fuertemente conexo** si existe un camino entre cualquier par de nodos que forman el grafo.
  - Si el grafo no es conexo, se pueden determinar las **componentes fuertemente conexas**




Grafo conexo




Grafo dirigido fuertemente conexo

# 6. Componentes conexas de un grafo

- **Carácter práctico del problema de la conexión:**
  - Red de comunicaciones: ¿puede una persona enviar un mensaje a cualquier otra persona?
  - Red de transporte: ¿es posible para todo vehículo desplazarse de una ubicación a cualquier otra ubicación?

- RESPUESTA: SÍ sólo si los grafos correspondientes son conexos (o fuertemente conexos si los problemas se modelan usando grafos dirigidos)

# 6. Componentes conexas de un grafo

Algoritmo para determinar las componentes conexas de un grafo G **no dirigido**:

1. Realizar un recorrido del grafo a partir de cualquier vértice *w*. Los vértices visitados se guardan en el conjunto W.
2. Si el conjunto W es el conjunto de todos los vértices del grafo, entonces el grafo es conexo.
3. Si el grafo es no conexo, W es una componente conexa.
4. Se toma un vértice no visitado, *z*, y se realiza de nuevo el recorrido del grafo a partir de *z*. Los vértices visitados Z forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices han sido visitados.

# 7. Componentes fuertemente conexas

- **Algoritmo para determinar las componentes conexas de un grafo G dirigido**

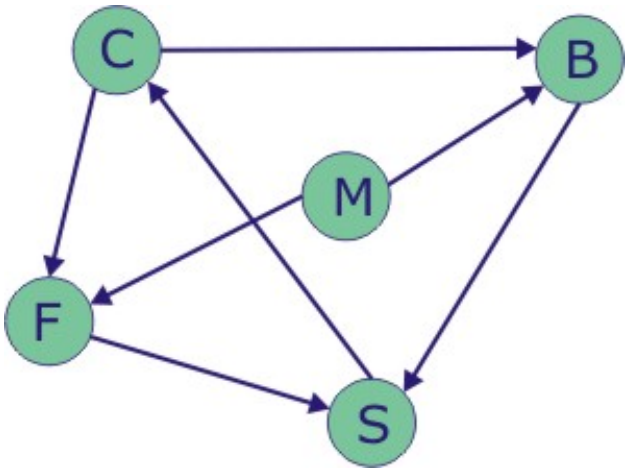
1. Obtener D(*v*)=conjunto de descendientes de un vértice de partida *v*, incluido el propio vértice *v*.
   - Así, tenemos todos los vértices desde los que hay un camino que comenzando en *v* llega a ellos.
   - Para ello se realiza un recorrido en profundidad del grafo G a partir del vértice *v*. Los vértices que son visitados se guardan en el conjunto D(*v*).

2. Obtener A(*v*)=conjunto de ascendientes de *v*, incluido el propio vértice *v*.
   - Estos vértices son aquellos en los que comienza un camino que llega a *v*.
   - Para ello se construye otro grafo $G_i$ que sea el resultado *de invertir las direcciones* de todos los arcos de G y a continuación se procede como en el paso 1.

3. $D(v) \cap A(v)$ es el conjunto de vértices de la componente fuertemente conexa a la que pertenece *v*.
   - Si $G = D(v) \cap A(v)$ entonces G es fuertemente conexo.
   - Si $G \neq D(v) \cap A(v)$ entonces G no es fuertemente conexo.
     - $D(v) \cap A(v)$ es una componente fuertemente conexa
     - Se selecciona un vértice cualquiera *w* que no esté en ninguna componente fuerte de las encontradas ($w \notin D(v) \cap A(v)$) y se repite a partir del paso 1 hasta obtener todas las componentes fuertemente conexas del grafo.

# 7. Componentes fuertemente conexas

- Ejemplo
  1. D(C)={C, F, S, B}
  2. A(C)={C, S, F, M, B}
  3. G ≠ D(C)∩A(C) ⇒ G NO es fuertemente conexo.
  4. El único vértice que no está en una componente fuertemente conexa es M, que forma otra componente fuertemente conexa.



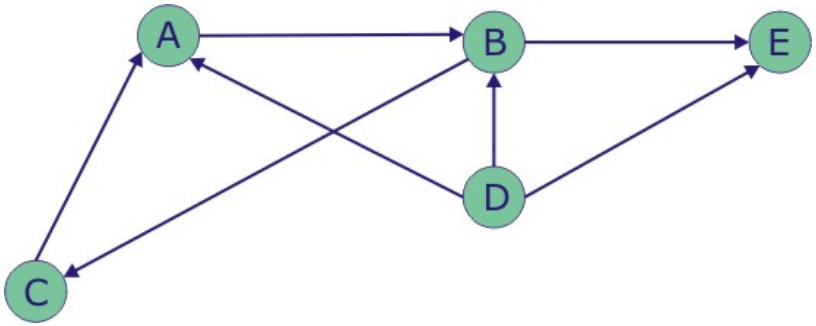

Componentes fuertemente conexas de G:

{C,F,S,B} y {M}

# 8. Matriz de caminos

- **Camino entre un par de vértices**
  - **G** grafo de *n* vértices y A su matriz de adyacencia de tipo lógico.
  - (A[i][1] AND A[1][j]) OR (A[i][2] AND A[2][j]) … OR (A[i][n] AND A[n][j])
    - TRUE (1) implica que existe al menos un camino de longitud 2 desde el vértice i al vértice j que pase a través del vértice 1, o del vértice 2, o del vértice *n*.
  - Si cambiamos AND por producto y OR por suma, la expresión representa el elemento $A_{ij}$ de la matriz $A^2$
    - Los elementos $A_{ij}$ de la matriz $A^2$ son TRUE si existe un camino de longitud 2 desde el vértice i al vértice j $\forall i,j=1,\ldots,n$
  - El producto matricial $A^2 x A$ nos permite determinar si existe un camino de longitud 3 entre cualquier par de vértices del grafo.
  - En general, para determinar la existencia de un camino de longitud *m* entre cualquier par de vértices se forma el producto booleano de los caminos de la matriz $A^{m-1}$ con la matriz de adyacencia A.

- **Camino entre un par de vértices**
  - Ejemplo:




Matriz de adyacencia A

$$A = \begin{pmatrix} F & T & F & F & F \\ F & F & T & F & T \\ T & F & F & F & F \\ T & T & F & F & T \\ F & F & F & F & F \end{pmatrix}$$

Producto Booleano AxA

$$A \times A = A^2 = \begin{pmatrix} F & F & T & F & T \\ T & F & F & F & F \\ F & T & F & F & F \\ F & T & T & F & T \\ F & F & F & F & F \end{pmatrix}$$

$A_{15}$=TRUE pues hay camino de longitud 2 de A a E (A→B→E)

Producto Booleano $A^2$xA

$$A^2 \times A = A^3 = \begin{pmatrix} T & F & F & F & F \\ F & T & F & F & F \\ F & F & T & F & T \\ T & F & T & F & T \\ F & F & F & F & F \end{pmatrix}$$

$A_{41}$=TRUE pues hay camino de longitud 3 de D a A (D→B→C→A)

- **Matriz de caminos**
  - Sea G un grafo con *n* vértices. La matriz de caminos de G es la matriz *n*×*n* **P** definida como:

$$P_{ij} = \begin{cases} 1 & \text{si hay un camino desde } v_i \text{ a } v_j \\ 0 & \text{si no hay camino desde } v_i \text{ a } v_j \end{cases}$$

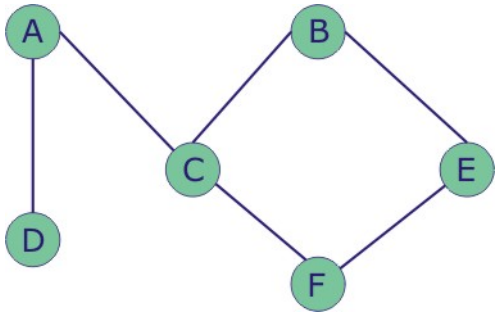
  - Relación entre P, A y las sucesivas potencias de A:
    - Dada la matriz $B_n = A^1 + A^2 + \ldots + A^n$, la matriz de caminos P:

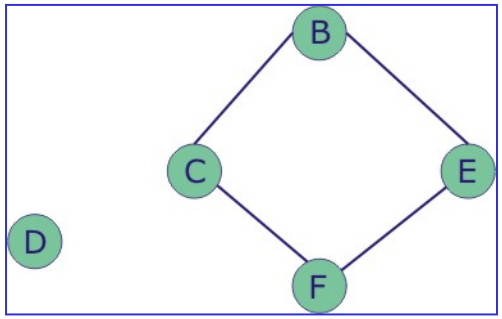
$$P_{ij} = \begin{cases} 1 & \text{si y sólo si } B_n(i,j) \geq 1 \\ 0 & \text{en otro caso} \end{cases}$$

  - Grafo fuertemente conexo
    - Si para todo par de vértices $v_i$, $v_j$ existe un camino de $v_i$ a $v_j$ y un camino de $v_j$ a $v_i$.
    - La matriz de caminos P ha de tener todos los elementos igual a 1 excepto la diagonal principal.

# 9. Puntos de articulación de un grafo

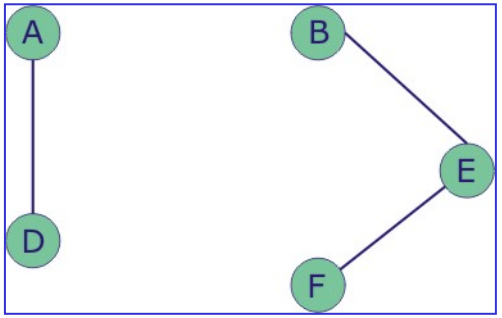
- **Punto de articulación de grafo no dirigido:**
  - Vértice V que tiene la propiedad de que si se elimina junto a sus arcos, la componente conexa en que está el vértice se divide en 2 o más componentes.
  - Ejemplo:
    - puntos de articulación (G)={A, C}
    - Si se elimina cualquier otro vértice, el componente conexo no se divide




Sin vértice A:

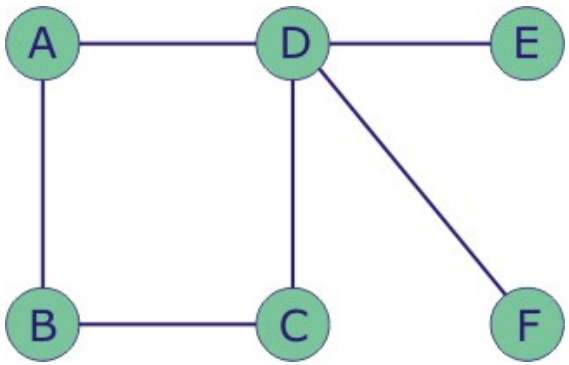
{C, B, E, F} y {D}

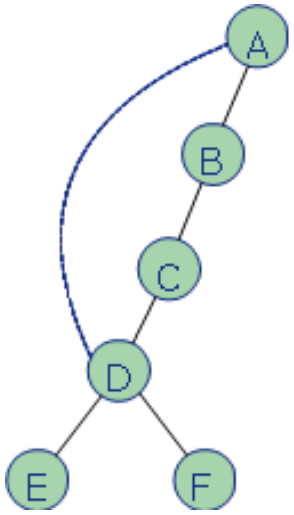



Sin vértice C:

{B, E, F} y {A, D}

- **Algoritmo búsqueda de puntos de articulación:**
  - Utiliza el recorrido en profundidad RECURSIVO del grafo
  - El recorrido recursivo en profundidad puede representarse mediante un ÁRBOL DE RECUBRIMIENTO.
    - La raíz del árbol es el vértice de partida A.
    - Cada arco del grafo estará como una arista en el árbol.
      - Al pasar por los vértices adyacentes de v, arcos(v,u):
        - Si el vértice u no está visitado, entonces (v,u) se dice que es una arista del árbol
        - Si el vértice u ya se ha visitado, entonces (v,u) se dice que es una arista hacia atrás

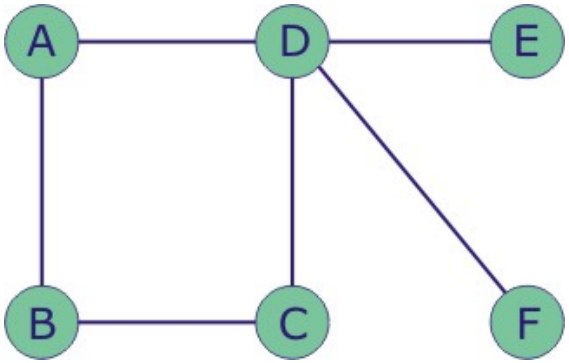



Numerando los nodos del árbol en preorden (RID) obtenemos el orden en que han sido visitados

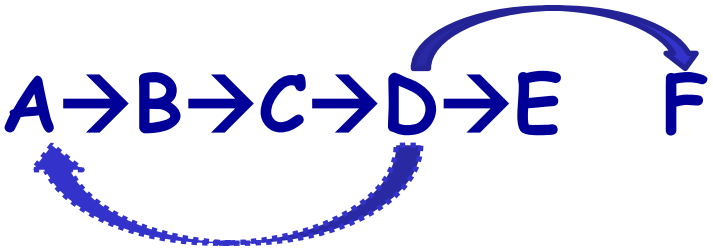
# 9. Puntos de articulación de un grafo

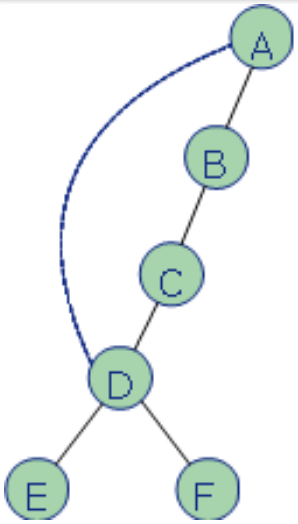
- El recorrido **recursivo** en profundidad puede representarse mediante un ÁRBOL DE RECUBRIMIENTO.




A →RR(A) {A} Adyacentes no visitados: {B,D}
→RR(B) {A,B} Adyacentes no visitados: {C}
→RR(C) {A,B,C} Adyacentes NV: {D}
→RR(D) {A,B,C,D} Adyacentes NV: {E,F}
→RR(E) {A,B,C,D,E} Adyacentes NV:{}
→RR(F) {A,B,C,D,E,F} Adyacentes NV:{}
→RR(D): YA VISITADO!! Arco hacia atrás
No hay nodos sin visitar, por lo que termina el recorrido:
{A,B,C,D,E,F}

Numerando los nodos del árbol en preorden (RID) obtenemos el orden en que han sido visitados



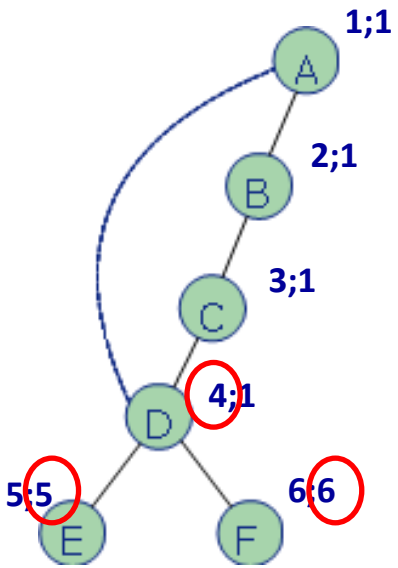
- **Algoritmo búsqueda de puntos de articulación de grafo conexo:**

1. Recorrer el grafo en profundidad a partir de cualquier vértice. Se numeran en el orden en que son visitados los vértices, esta numeración la llamamos *Num(v)*.
2. Para cada vértice *v* del árbol del recorrido en profundidad determinamos el vértice de numeración más baja (*Bajo(v)*) que es alcanzable desde *v* a través de 0 ó más aristas del árbol y como mucho una arista hacia atrás.
   - ***Bajo(v)*=mínimo de 3 valores:**
     - ***Num(v)***
     - **El menor valor de *Num(w)* para los vértices *w* de las aristas hacia atrás *(v,w)* del árbol.**
     - **El menor valor de *Bajo(w)* para los vértices *w* de las aristas *(v,w)* del árbol.**

# 9. Puntos de articulación de un grafo

- **Algoritmo búsqueda de puntos de articulación de grafo conexo:**

3. Determinación de los puntos de articulación:
   1. Raíz: (vértice de partida) punto de articulación si y sólo si tiene dos hijos.
   2. Cualquier otro vértice *w* es punto de articulación si y sólo si *w* tiene al menos un hijo *u* tal que *Num(w)<=Bajo(hijo)*




- Num(v)=recorrido preorden
- Bajo(v)=min{Num(v),Num(arista_atrás),Bajo(hijos)}

Como necesito Bajo(hijos), comienzo por las hojas:

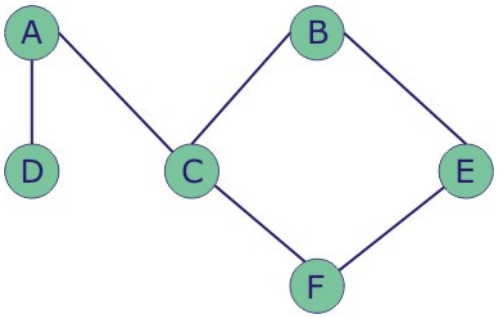
- Bajo(E)=min{Num(E),Num(arista_atrás),Bajo(hijos)}=5
- Bajo(F)=min{Num(F),Num(arista_atrás),Bajo(hijos)}=6
- Bajo(D)=min{Num(D),Num(A),Bajo(E),Bajo(F)}=min{4,1,5,6}=1
- Bajo(C)=min{Num(C),Num(arista_atrás),Bajo(D)}=min{3,1}=1
- Bajo(B)=min{Num(B),Num(arista_atrás),Bajo(C)}=min{2,1}=1
- Bajo(A)=min{Num(A),Num(arista_atrás),Bajo(B)}=min{1,1}=1

Sólo D cumple la condición, pues Num(D)<=Bajo(algún hijo)

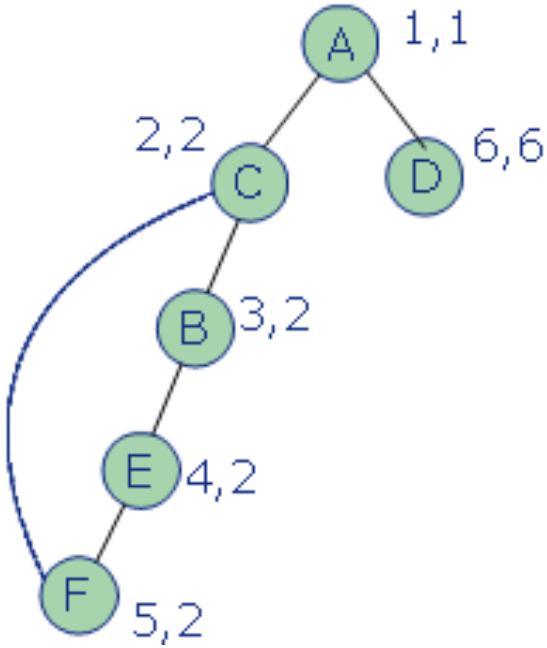
Se cumple para los dos hijos: para E: 4<=5 y para F: 4<=6

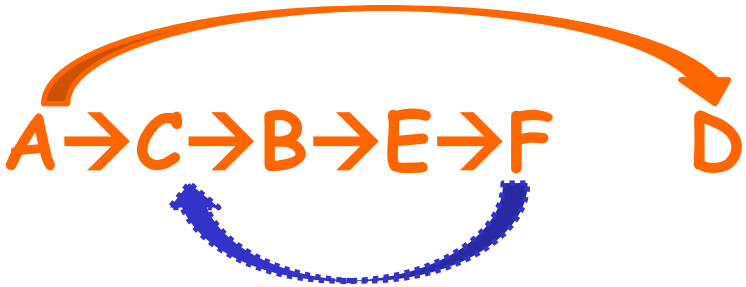
# 9. Puntos de articulación de un grafo

- Ejemplo:



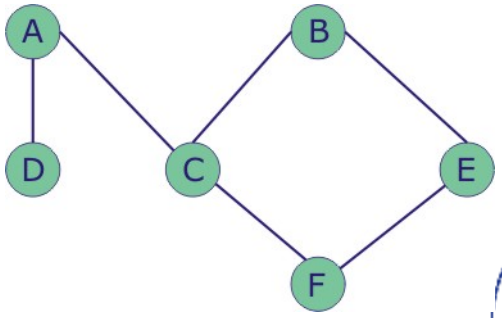

A →RR(A) {A} Adyacentes NV: {C,D}
→RR(C) {A,C} Adyacentes NV: {B,F}
→RR(B) {A,C,B} Adyacentes NV: {E}
→RR(E) {A,C,B,E} Adyacentes NV: {F}
→RR(F) {A,C,B,E,F} Adyacentes NV:{}
→RR(F): YA VISITADO. ARCO HACIA ATRÁS a quien lo llamó: C
→RR(D) {A,C,B,E,F,D} Adyacentes NV:{}
No hay nodos sin visitar, por lo que termina el recorrido:
{A,C,B,E,F,D}

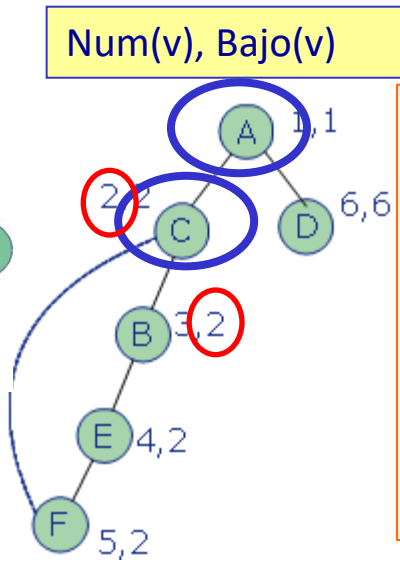


# 9. Puntos de articulación de un grafo

- **Ejemplo:**




Bajo(F)=min {Num(F), Num(C),Bajo(hijos)}= min{5,2}=2
Bajo(E)=min{Num(E),Num(arista_atrás),Bajo(F)]=min{4,2}=2
Bajo(B)=min{Num(B),Num(arista_atrás),Bajo(E)] =min{3,2}=2
Bajo(C)=min[Num(C),Num(arista_atrás),Bajo(B)]=min{2,2}=2

Bajo(D)=min{Num(D), Num(arista_atrás), Bajo(hijos)}=6

Bajo(A)=min{Num(A),Num(arista_atrás),Bajo(C),Bajo(D)]=
min{1,2,6}=1

- Num(v)=posición en recorrido en profundidad: {A,C,B,E,F,D}
- Bajo(v)=mín de:
  - Num(v)
  - min(Num(w)), siendo (v,w) arista hacia atrás
  - min(Bajo(w)) siendo (v,w) arista
- Puntos articulación:
  - **A**: raíz con nº hijos ≥ 2
  - **C**: Tiene hijo B con Num(C)<=Bajo(B)